Concurrent Execution of Mutually Exclusive Alternatives

Jonathan M. Smith

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY
1989

Copyright © 1989
Jonathan Michael Smith
All Rights Reserved

Abstract

Concurrent Execution of Mutually Exclusive Alternatives

Jonathan M. Smith

We examine the task of concurrently computing alternative solutions to a problem. We restrict our interest to the case where only one solution is needed: in this case we need some rule for selecting between the solutions. We use ``fastest first,'' where the first successful alternative is selected. For problems where the required execution time is unpredictable this method can show substantial execution time performance increases. These increases are dependent on the mean execution time of the alternatives, the fastest execution time, the overhead involved in concurrent computation, and the overhead of selecting and deleting alternatives. Rather than using the traditional approach of multiple computers cooperating on the solution to a problem, this method achieves a solution competitively.

Among the problems with exploring multiple alternatives in parallel are side-effects and combinatorial explosion in the amount of state which must be preserved. These are solved by process management and an application of ``copy-on-write'' virtual memory management. The side effects resulting from interprocess communication are handled by a specialized message layer which interacts with process management.

We show how the scheme for parallel execution can be applied to several application areas. The applications are distributed execution of recovery blocks, OR-parallelism in Prolog, and polynomial root-finding.

Table of Contents

	Page
1. Introduction	1
1.1. Thesis Summary	4
2. Theoretical Underpinnings	8
2.1. Extremal Behavior	8
2.2. Performance Analysis	10
2.2.1. Overhead	11
2.2.2. Analytic Description	11
2.2.3. Parallel Speedup	14
2.2.4. Domain-wide performance indices	19
2.3. Conclusions	21
3. Algorithms for Parallel Execution	22
3.1. System Model	22
3.2. Process Management	23
3.2.1. Synchronization	25
3.2.2. Atomicity	26
3.3. Predicates	26
3.3.1. Representation of Predicates	27
3.4. Interprocess Communication	32
3.4.1. Messages	32
3.4.2. Multiple Worlds	33
3.5. Discussion	37
4. Implementation, Applications and Experiments	40
4.1. Measurement of Overhead Costs	40
4.2. Copy-on-write	41
4.2.1. Motivation	42
4.2.2. Data Acquisition	42
4.2.3. Data Analysis	44
4.2.4. Relationships	50
4.2.5. Write Fraction for Real Programs	54
4.2.5.1. Franz Lisp	55
4.2.5.2. GNU Emacs	57
4.2.6. Conclusions about copy-on-write	58
4.3. Remote *fork()*	60
4.3.1. Further process migration ideas	60
4.4. Disk Response Time	62
4.5. Network Response Time	63
4.6. Sibling Elimination	64
4.6.1. Real Time	66
4.6.2. System Time	74
4.6.3. Real Time, 16 Procs only	82
4.6.4. System Time, 16 Procs only	88
4.6.5. Correction for process scheduling	94

4.7. Possible Sources of Error in Measurements 100
4.7.1. UNIX Clock Facility 101
4.7.2. Experimental Apparatus 102
4.8. Discussion 104
4.9. Applications 104
4.9.1. Distributed Execution of Recovery Blocks 104
4.9.2. Polynomial Root-finding 106
4.9.2.1. Example 107
4.9.2.2. Parallel Execution 113
4.9.3. Other Applications of the Technique 117
4.9.3.1. OR-parallelism in *Prolog* 117
4.9.3.1.1. Tutorial Example 118
4.9.3.1.2. Existing Solutions 120
4.9.3.1.3. Discussion 126
4.9.3.1.4. Measurements of Published *Prolog* Programs 128
4.9.3.2. Polyalgorithms 130
4.9.3.3. Simulation 131
5. Related Work 132
6. Conclusions 135
7. Directions for Future Work 141
8. References 142
9. Appendix I: *do_fork.c* 153
10. Appendix II: netrand.c 155
11. Appendix III: Further Jenkins-Traub executions 157
11.1. Polynomial #1 157
11.2. Polynomial #2 158
11.3. Polynomial #3 160
11.4. Polynomial #4 161
11.5. Polynomial #5 163
11.6. Polynomial #6 164
11.7. Polynomial #7 165
11.8. Polynomial #8 166
12. Appendix IV: Source, Prolog Sorts 168
13. Appendix V: Timings, Prolog Sorts, Large Lists 170
14. Appendix VI: Sort Performance, Small Lists 173
15. Appendix VII: Naive Sort Performance 174
16. Appendix VIII: Performance on Small, Ordered Lists 175
17. Appendix IX: Program to estimate memory speeds 176
18. Appendix X: Lower bound affects dispersion 178
19. Appendix XI: do_elim Script 184
20. Appendix XII: do_elim.c 186
21. Appendix XIII: "C" version of Jenkins-Traub algorithm 194
22. Appendix XIV: cvaryangle.c 208
23. Appendix XV: r2p.c 210
24. Appendix XVI: cmach.c 215
25. Appendix XVII: Extremal exploitation of randomness 220
26. Biography 224

Acknowledgments

First and foremost, I'd like to thank my advisor, Gerald Quentin Maguire, Jr. (''Chip''), who was a constant source of ideas and constructive criticism, was a collaborator in much of the experimental work reported here, and is a great friend as well, beyond his role as my thesis advisor.

Robert Strom has been extremely helpful in my gaining an understanding of the problems, approaches, and trade-offs; he has inspired many of the ideas I've presented here. Rob also served as an unofficial member of the dissertation committee.

Discussions with Calton Pu, Yechiam Yemini, Steven Feiner and David Farber have contributed to this thesis. Further useful suggestions were made by Mischa Schwartz and Martin Vetterli in the process of defending the thesis. Salvatore Stolfo pointed out *Prolog* OR-parallelism, and Joseph Traub suggested numerical polyalgorithms as applications. John Ioannidis was co-implementor of the *rfork()* mechanisms. David Presotto pointed me towards recovery blocks. Colin Harrison of the IBM T.J. Watson Research Center kindly allowed the use of an experimental multiprocessor for the performance measurements.

I'd also like to thank my parents and grandparents for their long years of support. Refen Koh and Rodney Farrow have given me many suggestions and much support.

This work was supported in part by equipment grants from the Hewlett-Packard Corporation and AT&T, Defense Advanced Research Projects Agency contract N0039-84-C-0165, and NSF grant CDR-84-21402.

UNIX and WE 32101 are registered trademarks, and 3B2 is a trademark of AT&T; HP-UX, HP9000, and HP are trademarks of the Hewlett-Packard Corporation. VAX, Digital, UNIBUS, UDA50, HSC50, RA81, and DEC are Trademarks of Digital Equipment Corporation.

1. Introduction

This thesis examines the problem of concurrently executing alternative solutions to a problem. There are four major questions which scientific research can answer. The first question is "can we do this?;" the second is "if so, how?," the third is "when do we want to?," and the fourth, "where will opportunities exist?."

This thesis answers each of these questions for this problem; a summary of the thesis results can be found at the end of the introduction.

Over time, the ratio of people to computers has decreased[1]. In the mainframe environments of the 1960s, there were often several thousand users per computer. As time advanced, departmental minicomputers which became popular in the 1970s had an order of magnitude fewer users for each computer, even though these computers were roughly comparable in power to the earlier generation. The 1980s have seen the introduction of supermicrocomputers where the number of users is a dozen or less, and workstations and personal computers which may support only a single user. Thus, it is reasonable to expect that in the near future, rich computational environments will offer use of several computers at the same time. These computers are typically connected through a bus, where the communications abstraction is a shared memory, or a network, where the communications abstraction is message-passing. A description of some computer organizations which incorporate multiple processors can be found in Smith [Smith1986a].

A question which has intrigued many researchers is how this increasing supply of computational resources, in the form of multiple computers, can be used to solve bigger problems, to solve problems faster, and to solve problems more reliably. Traditional approaches have been *cooperative,* in the sense that the multiple computers cooperate in developing the problem solution. Here, we look at a *competitive* approach, that of pursuing alternative solution methods.

There are many situations where there exist several alternative methods for computing a result, where a result in the most general case is a state change. Our designs show what can be done to execute instances of this problem type, *speculatively,* in parallel.

We are interested in what performance gains can be achieved. We measure

[1] These are observations noted by Nelson [Nelson1987a] in a videotaped lecture.

performance using the metric of execution time, which is the amount of wall clock time necessary to carry out a computation. Thus, we may increase performance by this measure, while decreasing performance by measures such as *throughput,* which is a measure of the amount of useful work done per unit time.

This thesis demonstrates that the speedup promised by parallel exploitation of randomness is achievable in practice.

We begin by describing the computations to be analyzed. These are essentially a set of alternative methods for causing a state change to take place, with the additional constraint that at most one alternative state change occurs. These might be denoted a, b, and c, and expressed as

```
SELECT
   a
 OR
   b
 OR
   c
TCELES
```

a, b, and c comprise a block. The block's semantics are simple: one of its components is executed.

Use of other language features allows control of the execution, if necessary. For example, if one wanted to execute the alternatives in order a, b, c, loops and guards could be applied. e.g.,

```
FOR I=1 TO 3
  SELECT
    WHEN I=1: a
   OR
    WHEN I=2: b
   OR
    WHEN I=3: c
  TCELES
ROF
```

Only alternatives with open guards can complete. Some of the alternatives may compute an acceptable result, while others may not. The essential problem is the choice between successful alternatives, or an indication of failure if there are no such alternatives. An

error condition is raised when no alternative is successful. An ALGOL-like language construct embodying this situation is:

```
ALTBEGIN
    ENSURE guard_1 WITH method_1 OR
    ENSURE guard_2 WITH method_2 OR
                 .
                 .
                 .
    ENSURE guard_N WITH method_N OR
    FAIL   /* no method succeeded */
END
```

Figure 1: Alternative Block

What we want is for at most one method to be applied to our problem, or for whatever conditions constitute failure to be indicated. Each method, $1..N$, has associated with it a *guard* condition, which it must satisfy to be considered successful. A method is called an *alternative*. When the alternatives are composed into a block, as illustrated in figure 1, the interpretation is that one alternative (possibly failure) is selected non-deterministically. The non-determinism defined in the semantics of alternative selection can often be exploited with parallelism for higher-performance computing. Non-deterministic algorithms have been discussed in the literature; Cohen [Cohen1979a] provides a good survey; he mentions the possibility of parallel execution, algorithms for which we will describe in Chapter 3.

The selection is non-deterministic and *unfair*, in that the selection of alternates is not equiprobable, and should not be; it's clear that the alternative of failure should be given as low a probability of success as is possible, noting that when all the alternatives fail its conditional probability must be 1. The semantics of the construct are similar to Dijkstra's [Dijkstra1976a] guarded commands, in the special case where the same guard is used for all the statements. In an implementation setting, the construct resembles the Ada [Ledgard1981a] `select` with guarded alternatives; the selection of open (i.e., have satisfied the guard) alternatives is arbitrary.

Once our model is defined, and the semantics thus fixed, we can apply semantics-

preserving transformations to increase performance or achieve other goals. A successful transformation, then, has two requirements. First, it must correctly preserve the semantics. Second, it must achieve the goal set for it, e.g., a performance increase.

We present (1) a model for selection of alternatives in a sequential setting, (2) a transformation which allows alternatives to execute concurrently, (3) a description of the semantics-preservation mechanism, and (4) parameterization of where the performance improvements can be expected. Additionally, we show example application areas for our method.

1.1. Thesis Summary

This thesis examines methods for solving problems for which there are alternative solution methods. When all such solution methods are equally acceptable, executing the alternatives concurrently and selecting the first successful computation can improve response time. The particular solution method which is fastest often depends on the data. The major contributions of this work are:

- A scheme for parallel execution of nondeterministic algorithms. Consistency and correctness are ensured with "Multiple Worlds," where "copy-on-write" page-managed storage prevents side-effects and inhibits combinatorial explosion in state storage requirements.
- An analysis of the possible speedup which identifies necessary properties of the alternatives.
- An analysis of the overhead involved in the proposed parallel execution scheme, tools for measuring this overhead, and measurements derived with these tools.
- Use of the *rfork()* remote fork mechanism to create a copy of a running process; this extends the idea of process migration.
- Use of *sibling elimination* to reduce the scheme's effect on throughput.
- *Predicates* to extend the scheme to allow interprocess communication, where interacting processes are also "copy-on-write."
- Example applications from three areas: parallel execution of logic programs, concurrent execution of recovery block alternates expressed in our language (RB), and numerical analysis, particularly polynomial root-finding.

The scheme for parallel execution (Chapter 3) relies on the availability of alternative solution methods. Nondeterministic algorithms give rise to alternatives by making several choices, each one of which gives rise to an alternative. These alternatives are executed in parallel; the results of the first successful computation are chosen. At this point, the slower computations can be eliminated. The alternatives share a common parent process, from which their state is inherited "copy-on-write" so that unchanged state can be shared, while internal consistency and correctness of each alternative are maintained. The *rfork()* primitive extends the spawning of alternatives to a distributed setting. To ensure correctness of interprocess communication, completion predicates are attached to all messages sent by a process which has siblings; these ensure that receiving processes do not make updates which depend on state which may be eliminated.

We identify the sources of overhead for concurrent execution using this scheme, and provide a measure of the performance improvement (*PI*) which can be used to compare execution strategies. We use this measure to identify opportunities for response time improvement. The best sort of situation (discussed formally in Chapter 2) for our approach is one where: the alternatives require a significant amount of computation time, as encapsulated in the mean of their execution times; each alternative changes a small amount of the state of the calling process, thus reducing the overhead due to copying state; there is enough difference between the execution times of the alternatives that choosing the fastest and killing the others is worth the overhead of spawning the copies and deleting the slower siblings. The speedup is dependent on both the fraction of execution time devoted to overhead and the variance exhibited by the execution times of the alternatives. The implication is that if overhead can be understood and controlled, there is an opportunity for speedup roughly proportional to the variance; thus superlinear speedups, where the execution time is less than $O(1/N)$ for N processors, are *possible* under this scenario.

Since the potential for speedup is sensitive to overheads, we examined these overheads in Chapter 4. Copying is the major overhead in the creation and maintenance of the concurrent executions. Although there had been no previous work examining the efficacy of "copy-on-write," our results indicate that the technique is extremely effective in practice. We provide several useful measurement tools to gather execution time data. These tools are used to adapt our measurements to new computers; thus, with a few

measurements the domain for performance improvement using this method can be delimited. We implemented a system which performs a *remote fork*, which is similar to *process migration* with the exception that two processes exist when the operation is successful, rather than one migrated process. Our measurements confirm that the major overhead cost is copying, and further, offer empirical proof that child processes can be "spawned" in a distributed execution environment. Response time is affected by the response times of several system components; we examine two subcomponents, the disk and the network, in chapter 4. The final overhead is *sibling elimination*, which we modeled on a multiprocess timesharing system. This should represent the worst case execution, for which the results are encouraging, as they show the elimination to be remarkably cheap, and insensitive to process behavior. Given that the overheads are predictable and parameterizable these parameters can be used in deciding whether to apply the concurrent execution scheme.

After identifying the opportunities for response time improvement with analytic work and measurements, some example applications for concurrent execution are described. These example applications are drawn from several different areas of computer science, and illustrate the general utility of the scheme.

- *Parallel implementation of logic programming languages*, particularly OR-parallelism in *Prolog*, provides an appropriate environment, because the computation is data-driven. The execution time and control flow can vary greatly with the input. The way in which unification operates (as a "sophisticated pattern matcher") leads to an overwhelming preponderance of read references made to page-managed memory; while a high percentage of references are writes, these references are mainly to the stack, and thus locality is high; stack "growth" can be handled locally, reducing copying. Many logic programs have a great degree of parallelism, so that appropriate opportunities must be identified with respect to the overheads implied by our scheme. In particular, coarse-grained parallelism is better to exploit than fine-grained parallelism at the level of overhead we have observed. Our scheme deals with side-effects other than variable binding, and can run efficiently on general-purpose hardware.
- *Distributed execution of recovery block alternates* uses the "fastest-first" behavior

in an attempt to find a rapid failure-free path through the computation. Recovery blocks are designed for fault-tolerance, thus, there may be further requirements beyond fast execution time; we describe our RB language and suggest modifications to the concurrent execution scheme to increase robustness. These changes increase the amount of state available in a system to facilitate recovery.

- *Polynomial Root-Finding* with the Jenkins-Traub algorithm for finding complex roots of polynomials with complex coefficients was chosen as the third example. The sequential algorithm chooses an angle at random in its search for a root; we choose multiple angles and execute the choices in parallel. This application has shown speedups of a factor of 3, and some examples demonstrate desirable properties of our execution scheme in the face of alternative failures.

In summary, we examined a problem setting, introduced and applied a parallel execution scheme for improved response time, and demonstrated application areas and methods for determining appropriate applications through measurement of overhead.

2. Theoretical Underpinnings

2.1. Extremal Behavior

Consider the n random variables $X_1, \ldots, X_n$, each representing the execution time of a computation. The *order statistics* are computed by permuting the values of X_i to form a new set of values $X_{(1)}, \ldots, X_{(n)}$ such that $X_{(i-1)} \leq X_{(i)} \leq X_{(i+1)}$. This ascending order implies that $X_{(n)} = \max_{i=1}^{n} X_i$ and $X_{(1)} = \min_{i=1}^{n} X_i$. $X_{(1)}$ can be interpreted as the random variable defined by the first event to complete. Then, the distribution of $X_{(1)}$, $F(t) = P(X_{(1)} \leq t)$ is given by $1 - \prod_{i=1}^{n} [1 - F_i(t)]$ where $F_i(t)$ is the distribution of X_i. The random variables are assumed to be independent. If the random variables are exponentially distributed each with parameter G_i, then $1 - F_i(t) = exp(G_i \cdot t)$. Thus, $X_{(1)}$ is also exponentially distributed with rate (parameter) $\sum_{i=1}^{n} G_i$. This means that the average value of $X_{(1)}$ is given by $\frac{1}{\sum_{i=1}^{n} G_i}$. In particular, if the random variables are identically distributed (all the G_i are the same), the average time to completion is given by $\frac{1}{n \cdot G}$ or, in other words, the average time to completion (min) is $\frac{1}{n}$ times the average execution time of a replica, $\frac{1}{G}$. Thus, the performance gain in the extreme is $O(\frac{1}{n})$, that is, linear in the number of processes. A short discussion of limit theorems for order statistics is available in Feller [Feller1970a] and a detailed treatise on the behavior of these extremes is given by Galambos [Galambos1987a]. While the exponential distribution eases analysis and leads readily to a linear speedup, there remains the possibility of arbitrary speedups, since the speedup is a function of the distribution. We should note as an aside that system throughput is not affected if the linear speedup occurs, since it is a measure of the amount of useful work performed in unit time. We compare n processors executing until first completes versus one executing a random instance. $<X>$ is the *expectation* of random variable X. The analysis shows that $n \cdot <X_{(1)}> = <X_i>$; thus, there is no decrease in system throughput at the limit, even though the work of $n-1$ processors might be interpreted as wasted.

Since the random variables must be independent for the analysis to hold, the execution times should be random. Two cases exhibiting this behavior are given in Appendices XVII and X. Appendix XVII shows a simple search problem. The experimental apparatus is given; the problem closely approximates many classical search problems where the input is random. For example, there is an obvious mapping between the random search problem and the problem of finding a name in a phone directory, given a number. Measurements are shown in figure 2:

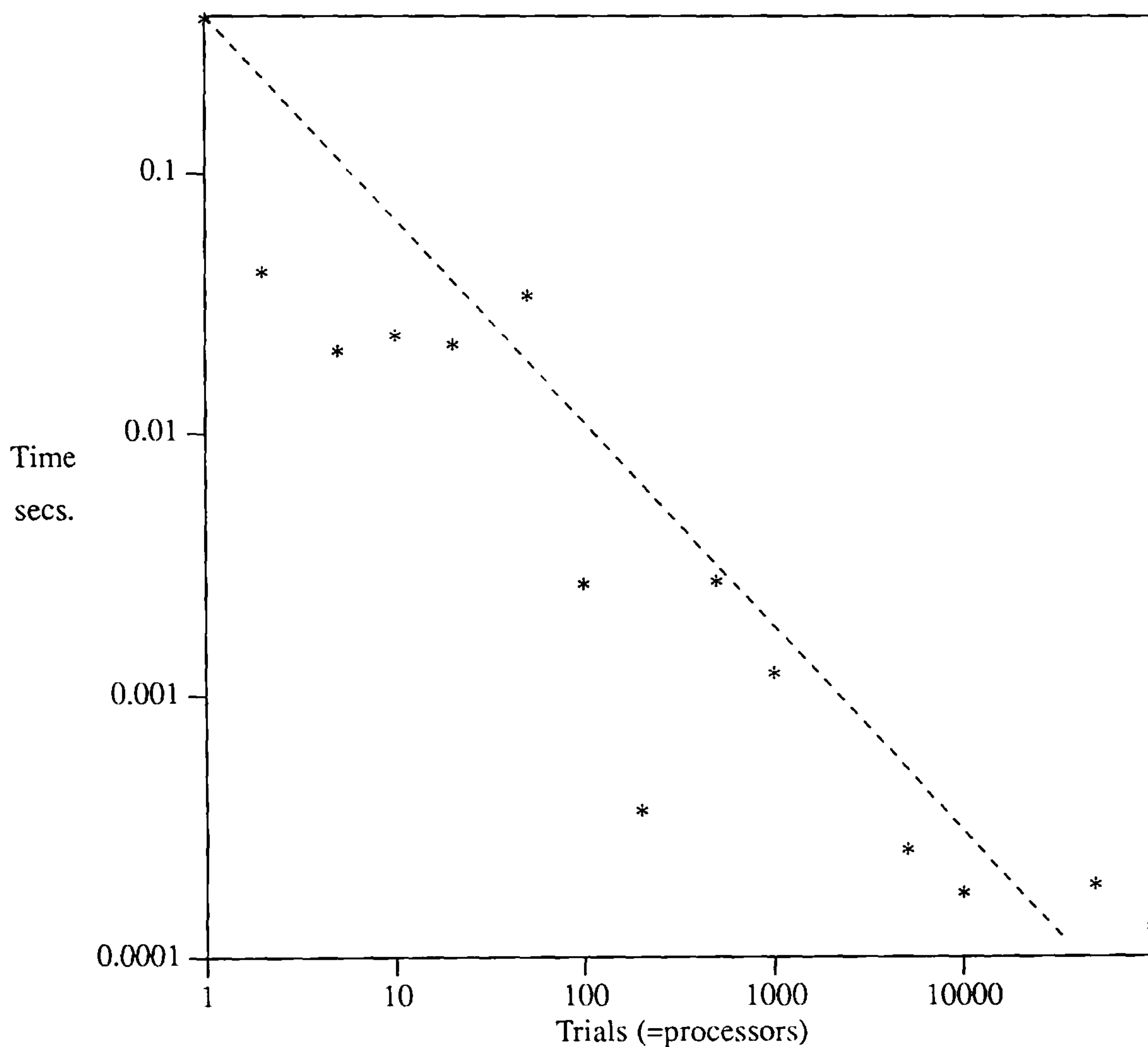


Figure 2: Use of randomness in search (Appendix XVII)

The results provide an existence proof that the tradeoff between randomization and parallelism can be exploited for speedups *in practice*. Another interesting aspect of our

approach is that the variance of the time to completion X is reduced by order $O(\frac{1}{n^2})$ when the execution times are exponentially distributed. This implies that the distribution of X tends to be very sharply focused around the best time possible. This fact has important implications for systems where the execution time must be predictable, such as real-time systems. Actual algorithms exhibit this behavior to the extent that you may randomize their execution times to be independent random variables. For example, Appendix X shows that a random choice of partitioning element in quicksort has little effect on the execution time, even though the behavior is as predicted, i.e., decreasing execution times and decreasing variance. The performance is limited in this case since the distribution function, $F()$, is bounded on the bottom by $O(n \log n)$, and the mean execution time of the algorithm approximates this quite closely.

The next section discusses the measurement of speedup we will use to test the success of the method, and shows the conditions necessary for this speedup to occur.

2.2. Performance Analysis

The possibility of a performance increase stems from the fact that we can select the fastest alternative by means of the synchronization protocol. The introduction's argument promises linear speedup. There are two facts which frustrate this promise. First, the dispersion of execution times must be significant, and the probability distribution function, $F()$, must have a long "tail." Second, we ignored the overhead involved in concurrent execution such as copying, sibling elimination, and processor contention.

The cost we must pay for obtaining execution time proportional to the time for the fastest alternate is use of available hardware.

Note that the action of continuing execution of the successful alternative and the process of sibling elimination can take place *asynchronously*. The effects of various overheads and system parameters are analyzed in the next section.

2.2.1. Overhead

To understand the overhead implied by the method, we should compare a sequential execution of the construct, in the best case, where the fastest alternative is selected. There are penalties we are paying for parallel execution of all alternatives. We must compare this scheme with sequential execution of the alternative which will be selected in any case. These penalties are:

1. Memory Copying. In the distributed case we must copy state for a remote child so that it can read or write locally. In the shared memory multiprocessor case, the copying overhead (in execution time) is reduced as the interprocessor bandwidth is much higher. There is additional copying to be performed during synchronization, as the changed state is updated in the parent's storage. The parent is constrained to remain blocked while the children are executing.

2. Sibling elimination. This is asynchronous, and naturally parallel, but the instructions to terminate the alternates must still be issued, and they increase with the number of alternates.

3. Effect on throughput, due to wasted work. As our bias has been towards execution time as a performance goal, we were willing to trade away throughput. Users may want to know what the tradeoffs are here, so the effect on system throughput should be analyzed.

2.2.2. Analytic Description

Assume that we have N alternative methods of performing a *computation*. A *computation* is a transformation from an input set (or Domain) to an output set (or Range); these sets consist of *state vectors*, intended to describe the relevant state of the world, i.e., the machine state. For Domain **D** and Range **R**, $\vec{x} \in \mathbf{D}$ is transformed via the computation into some $\vec{y} \in \mathbf{R}$, thus we could write $\vec{y} = C(\vec{x})$ There may be several such C which we classify as interesting. Transformations of C which add or remove useless operations are infinitely numerous, but not interesting. *Algorithmic* differences, random parameters, or significant differences in implementation technique are interesting. Assume that the N alternatives postulated earlier are N such interesting Cs, and that they will be applied to some $\vec{x} \in \mathbf{D}$. Each C consists of some series of *steps*, where $\vec{x}$ is transformed into $\vec{x}'$, $\cdots$ until $\vec{y}$ is achieved. Each step requires some amount of clock time, τ, to complete; for

$C(\vec{x})$ $\tau(C,\vec{x})$ is the sum of these times. τ, the *execution time*, gives us a way of comparing the performance of two computational methods on the same input, say $\vec{x}$.

There are many practical situations in which we want to minimize the computation time required for the transformation of $\vec{x}$ to $\vec{y}$. We will denote the N alternatives as $C_1, \ldots, C_N$. Since our goal is minimizing execution time, let us consider some possible relations between the C_i on elements of **D**.

1. $\tau(C_i,\vec{x}) \leq \tau(C_j,\vec{x})$ for every $\vec{x} \in \mathbf{D}$ which interests us. It's clear that we should use C_i and discard C_j for every i and j for which this holds.
2. $\tau(C_i,\vec{x}) \leq \tau(C_j,\vec{x})$ for some $\vec{x}$ which interest us, and we can accurately predict for which $\vec{x}$ this relation holds. In this case, we can construct a synthetic computation, C_{N+1}, which selects C_i when this holds. To anchor the relation with an example, consider the case of two list-sorting algorithms, `Q` and `I`. `Q` is faster than `I` when the number of elements to be sorted is greater than 10. Thus, using this knowledge, we can construct a synthetic sorting routine as follows:

```
sort( list, size ) :=
    if( size > 10 )
        Q( list, size )
    else
        I( list, size ).
```

The synthetic routine partitions the input domain by performance, and thus achieves performance superior to either `Q` or `I`. The tough point here is the partitioning; it's rarely as simple to delimit performance boundaries as "`size < 10`." If the input set can be partitioned, but only at significant computational cost, the desired property of the synthetic routine, that $\tau(C_{N+1},\vec{x}) = \min_i \tau(C_i,\vec{x})$ for all $\vec{x}$ of interest, may be achievable with the following technique.

If all interesting $\vec{x}$ are known in advance, we can associate one C_i with each $\vec{x}$ in a precomputed table. Then, $\tau(C_{N+1},\vec{x})$ can be calculated by adding the cost of a table lookup to the cost of executing the table element on $\vec{x}$. In rare cases (rare due to the amount of state under consideration, and the sizes of **R** and **D** for the problems we are

interested in), we may be able to store $\vec{y}$ in the table, thus removing any execution time cost except for a table lookup.

3. $\tau(C_i,\vec{x}) \le \tau(C_j,\vec{x})$ for some $\vec{x}$ which interest us, but while interesting, the $\vec{x}$ cannot easily be related to $\tau(C_i,\vec{x})$ Essentially, this means that the table lookup technique cannot be used, because we cannot reasonably precompute the values of $\tau(C_i,\vec{x})$. This might be due to the input set, e.g., infinite size. For example, a naive *quicksort* is not stable, and where the list is *ordered* the sort is slow. In these cases, a stable sort with good performance, e.g., *heapsort*, may be preferable. However, it's clear that storing a lookup table of all "interesting" lists is infeasible, and pretesting for the "ordered" property is potentially expensive. Another problem is that $\tau(C_i,\vec{x})$ may vary due to the execution environment (which may or may not be described by $\vec{x}$; it probably should be, for completeness), e.g., processor type, multiprocessing workload, or interactions with other computations. In these cases, where performance on the $\vec{x} \in \mathbf{D}$ is unpredictable, we might try other schemes:

   A. Statistical data can be applied, e.g., *quicksort* is "almost always" $O(n\log n)$. Thus, we'll rarely go wrong using it.

   B. An algorithm can be selected at random from amongst the C_i when given $\vec{x}$.

   C. The C_i can be applied to $\vec{x}$ concurrently; the first C_i which produces $\vec{y}$ is selected. The other C_i are irrelevant and can be terminated. There is, however, overhead in setup and synchronization (selection) which cannot be ignored.

   Scheme **A** relies on information which may not be available. Scheme **B**, when run repeatedly on some uniformly distributed input $\vec{x}$, will perform at the arithmetic means of the computations' performance, i.e., $\frac{\sum_{i=1}^{N} \tau(C_i,\vec{x})}{N}$. Scheme **C** offers some opportunity for achieving the best performance on each input $\vec{x}$. We will try to characterize this opportunity. Note that there are two possibilities for concurrent execution, *real* and *virtual*. Real concurrency means that the evaluation of $C_i(\vec{x})$ is taking place simultaneously with that of $C_j(\vec{x})$ virtual means that there is some sharing of hardware, for example through multiprocessing.

2.2.3. Parallel Speedup

Our analysis must begin with semantics, as otherwise we are subject to criticism of the ``apples and oranges'' type. Such criticism stems from the observation that changing the problem in order to apply a program transformation makes performance results incomparable; we are comparing unlike programs.

To an observer, the concurrent execution of the C_i must look like Scheme **B.** (as discussed above); that is, that we have followed a single thread of computation, chosen arbitrarily from amongst $C_1, \ldots, C_N$. Since the $C_1, \ldots, C_N$ may update shared state described by $\vec{x}$, we solve the problem by copying state when needed and by selecting some C_i by virtue of its state changes. Thus, since the observer sees non-deterministic selection of one alternative, we must compare concurrent execution to sequentially performing some C_i, chosen arbitrarily (we'll assume randomly). Since, as stated previously, execution time is our figure of merit, we'll analyze with that intent, ignoring measures such as throughput. Arbitrary selection is trivial to implement; it costs no execution time for purposes of our analysis. The execution of the selected alternative costs $\tau(C_i,\vec{x})$ for the $\vec{x}$ under study. Thus, we can expect the mean cost to be $\frac{\sum_{i=1}^{N} \tau(C_i,\vec{x})}{N}$, the average of the C_i's times when applied to $\vec{x}$. For notational convenience, define C_{mean} such that

$$\tau(C_{mean},\vec{x}) = \frac{\sum_{i=1}^{N} \tau(C_i,\vec{x})}{N}$$

By executing the C_i concurrently, we will expect the cost of execution to be

$$\tau(C_{best},\vec{x}) + \tau(overhead)$$

where

$$\tau(C_{best},\vec{x}) \leq \cdots \leq \tau(C_{worst},\vec{x})$$

and *overhead* is complex. *Overhead* consists of operations performed to support concurrent execution which would not be necessary in the nondeterministic sequential case. It consists of the following components:

setup: Instead of simply calling C_i, we must now spend cycles creating execution

environments for $C_1, \ldots, C_N$: for example, setting up process table entries and page map tables.

runtime: This consists of copying memory areas which are shared between the $C_1, \ldots, C_N$ when updates are attempted. This performance is strongly influenced by *locality of reference*. Additionally, if C_{best} is sharing resources, e.g., CPU time, with some C_i, $i \neq best$, then all such C_i's runtimes must be added to the runtime overhead of C_{best}, as cycles spent processing C_i are not spent processing C_{best}.

selection: This is the cost involved in selecting C_{best}, e.g., deleting C_i *such that* $i \neq best$, cleaning up system state, such as performing the updates made by C_{best}, e.g., writing checks or bottling beer.

Thus, for a given $C_1, \ldots, C_N$ and $\vec{x}$,

$$\begin{aligned}
&\tau(overhead) = \\
&\quad \tau(setup(C_1 \cdots C_N, \vec{x}) + \\
&\quad \tau(runtime(C_{best}, \vec{x}) + \\
&\quad \tau(selection(C_{best}, C_1, \ldots, C_N, \vec{x}) \;) \;,
\end{aligned}$$

and the parallel execution wins at $\vec{x}$ *iff*

$$\tau(C_{best}, \vec{x}) + \tau(overhead) < \tau(C_{mean}, \vec{x}).$$

Thus, we can calculate the performance improvement (*PI*) as:

$$PI = \frac{\tau(C_{mean}, \vec{x})}{\tau(C_{best}, \vec{x}) + \tau(overhead)}$$

essentially a ratio of execution times. For illustration, consider a case where N=3, on input $\vec{x}$. Thus, we have three methods C_1, C_2, and C_3. Let $\tau(overhead)$ be 5. Some possible relations are tabulated:

	$\tau(C_1,\vec{x})$	$\tau(C_2,\vec{x})$	$\tau(C_3,\vec{x})$	PI
(1)	10	20	30	1.33
(2)	1	19	106	7.0
(3)	20	20	20	0.8
(4)	1	2	3	0.33
(5)	115	120	125	1.0
(6)	100	200	300	1.9

What can we infer from the examples? (3) indicates, along with (5), that the size of the differences matters. (4) shows that the relative magnitudes of the execution times and the overhead matters. (6) shows that the effects of the overhead (under our assumptions) diminish with increasing relative execution time. (2) illustrates a good situation, where the difference

$$\tau(C_{worst},\vec{x}) - \tau(C_{best},\vec{x})$$

is large. This magnitude of difference is well-encapsulated by such a statistical measure of dispersion [Robbins1975a] (letting values of τ serve as the random variable) as the *variance*, and the variance is easily computed as:

$$variance = mean\ value(\tau(C_i,\vec{x})^2) - PI^2 \cdot (\tau(C_{best},\vec{x}) + \tau(overhead))^2$$

However, this re-expression seems to serve little purpose, as we can manipulate the simple relationships describing PI into forms which genuinely ease analysis. Additionally, the variance, although a good measure of dispersion is not precisely what we want to predict speedup. This is because values other than the best execution time can be altered with no effect on the mean execution time. Altering these values can increase the value of the variance but not change the potential for speedup. In the remainder of this thesis, when we use the term ''variance'', we will mean the layman's *variance*, a measure of dispersion, rather than the statistician's *variance*.

We can analyze precisely the domains in which there is a performance improvement ($PI>1$). Letting $R_\mu = \frac{\tau(C_{mean},\vec{x})}{\tau(C_{best},\vec{x})}$ and $R_o = \frac{\tau(overhead)}{\tau(C_{best},\vec{x})}$, we can calculate PI as:

$$PI = \left[\frac{1}{1+R_o} \right] \cdot R_\mu$$

This re-expression isolates the effect of the dispersion, encapsulated in R_μ , from the effect of the overhead, encapsulated in R_o. Holding one of R_μ or R_o fixed allows us to estimate the effects on PI caused by the other. The behaviors are illustrated in figures 3 and

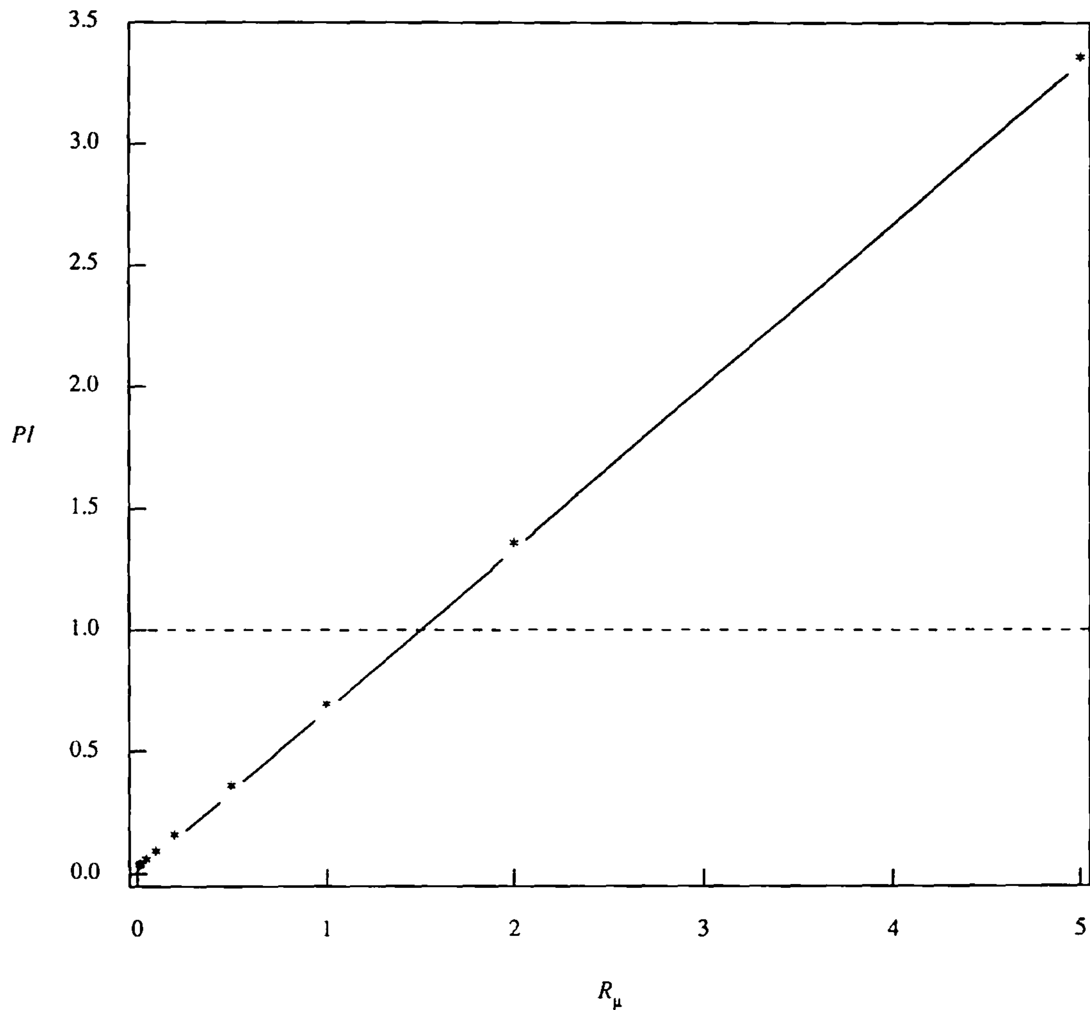


Figure 3: PI as a function of R_μ (R_o=0.5)

4. The relationship illustrated by the first figure is with R_o set to the constant value[2] 0.5;

[2] As reported in [Smith1988a] we observed a *write fraction*, which describes the fraction of memory copied by "copy-on-write" mechanisms, to be between 0.2 and 0.5. Thus 0.5 is reasonable, since the major overhead we observed was copying.

R_μ is varied between 0 and 5, and the values can easily be scaled. The curve is not interesting, as it's a direct proportion for fixed R_o; R_o determines the slope of the line, with R_o=0 the best case giving a slope of 1. This tells us that the performance improvement we can expect will be proportional to the variance of $\tau(C_i,\vec{x})$ damped by whatever effect $\tau(overhead)$ exhibits. Holding R_μ fixed and varying the overhead is more interesting, as figure 4 illustrates. The *y* axis has *PI* scaled proportional to $R_\mu = exp\,(1.0)$, and the scales are log-log to view a wide range of values.

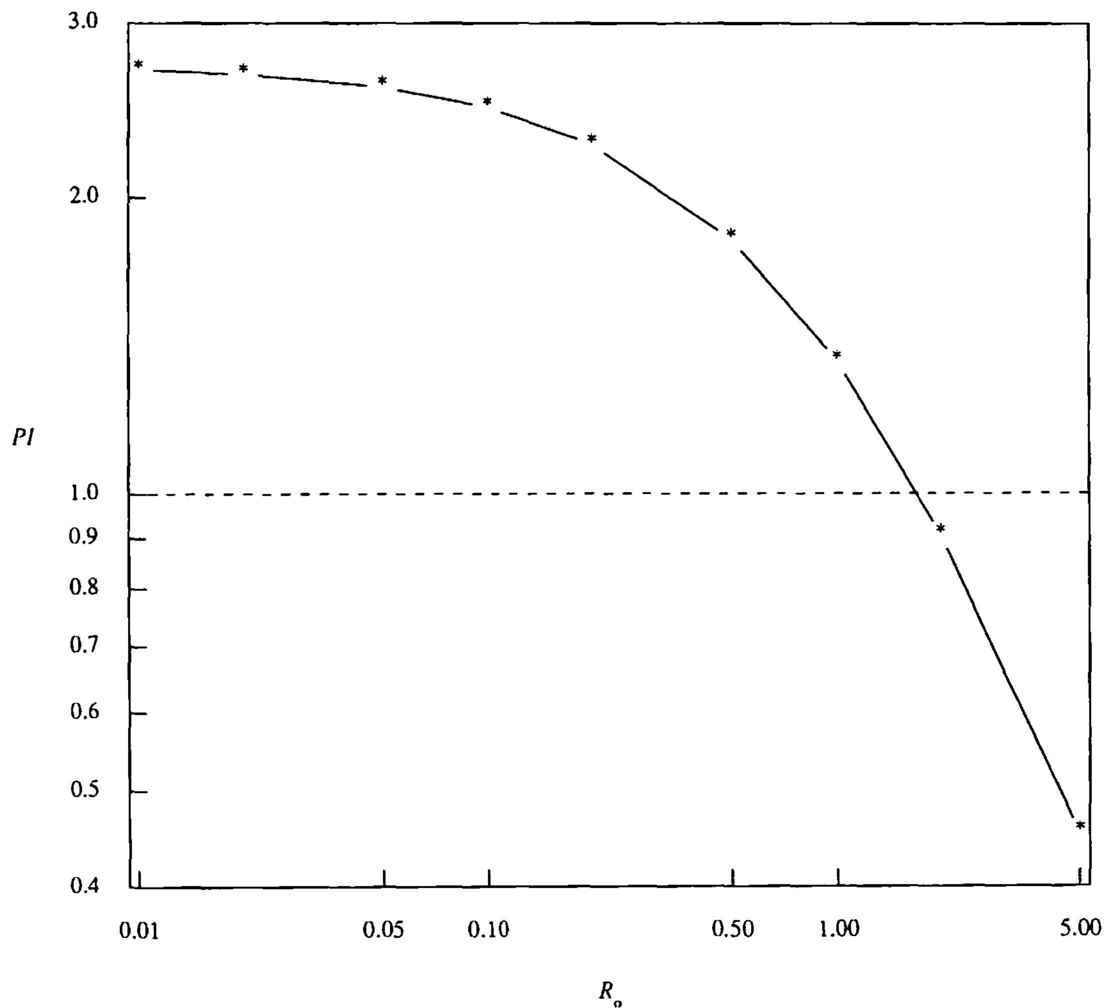


Figure 4: *PI* as a function of R_o ($R_\mu = exp(1.0)$)

This tells us that varying the overhead has a significant effect on the performance improvement we achieve, when scaled against the variance in execution times. An important fact which we can deduce from this performance analysis is that with sufficient variance, and small enough overhead, N processors can exhibit superlinear speedup by

parallel execution of N serial algorithms, as opposed to parallel execution of one serial algorithm which has been "parallelized."

2.2.4. Domain-wide performance indices

In the previous section, we constrained our analysis of the performance of concurrent execution to the performance at a single input state, encapsulated symbolically in $\vec{x}$. Thus, we have in our performance index, PI, a function of $\vec{x}$. This allows us to identify states at which the execution of alternatives will provide a performance improvement. Now, were we to consider the entirety of the input domain **D**, we might parameterize the performance at each $\vec{x}$ as $PI(\vec{x})$ .Our goal in securing a performance improvement is maximizing the "overall" performance improvement. How might we measure this? Let the overall performance improvement be calculated as

$$OPI(\mathbf{D}) = \sum_{\vec{x} \in \mathbf{D}} PI(\vec{x}) \cdot prob(\vec{x})$$

$Prob(\vec{x})$ is the probability that a given input state $\vec{x}$ will occur; we assume that $1 = \sum_{\vec{x} \in \mathbf{D}} prob(\vec{x})$ .We note that special care must be taken when $|\mathbf{D}|$ is $+\infty$, i.e., the input set is infinitely large, as (1) this case is common; and (2) we want a useful measure. Thus, our formulation of OPI serves as a weighted value of the $PI(\vec{x})$ values over $\vec{x}$ selected from domain **D**. This is just the *expectation* of $PI(\vec{x})$ $\langle PI(\vec{x}) \rangle$. If we assume the $\vec{x}$ are equiprobable, then

$$OPI = \sum_{\vec{x} \in \mathbf{D}} \frac{PI(\vec{x})}{|\mathbf{D}|}$$

since $prob(\vec{x})$ is $\frac{1}{|\mathbf{D}|}$. How good a measure is this? First, examine the case where there is no improvement in the performance, i.e. PI=1. Then,

$$OPI = \frac{\sum_{\vec{x} \in \mathbf{D}} 1}{|\mathbf{D}|} = 1,$$

which certainly makes sense, given the situation that this symbolic representation describes. If we expand the expression of $PI(\vec{x})$ we get

$$OPI(\mathbf{D}) = \sum_{\vec{x} \in \mathbf{D}} \left[\frac{\tau(C_{mean}, \vec{x})}{\tau(C_{best}, \vec{x}) + \tau(overhead)} \right] \cdot prob(\vec{x})$$

The value of $OPI(\mathbf{D})$ is maximized when the weighted average of the $PI(\vec{x})$ values is maximized. What factors are there? First, there is the input distribution, which for many interesting problems we know little about[3]. Second, there is the overhead, and the dispersion of values exhibited by $\tau(C_i,\vec{x})$ at a given $\vec{x}$. It's clear that reducing the overhead *always* helps; for the limiting case of $\tau(overhead) = 0$,

$$PI(\vec{x}) = R_\mu = \frac{\tau(C_{mean},\vec{x})}{\tau(C_{best},\vec{x})} \geq 1$$

Now, any dispersion of values of $\tau(C_i,\vec{x})$ causes $\tau(C_{mean},\vec{x}) > \tau(C_{best},\vec{x})$ so that $R_\mu > 1$, and thus $PI > 1$.

It is worthwhile to discuss the value of $prob(\vec{x})$ further. We estimated the sequential performance by assuming that a random choice be made from among the alternatives. This represents the ``best guess'' that could have been made using our statement of the problem. What if we relax the constraints on information available, so that rather than equiprobable inputs, we have some data gathered about the probability distributions of the input. The data may have been gathered through a small statistical sample, such as might be possible from a short historical record of the inputs[4]. How can this data help?

If $\tau(C_i,\vec{x}) < \tau(C_{mean},\vec{x})$ with probability $P_i > \frac{1}{|\mathbf{D}|}$, the expected performance of the sequential algorithm *might* be improved by *always* choosing C_i rather than choosing at random. Unfortunately, we also need to know the values of $\tau(C_i,\vec{x})$ to be effective. The difference in expected values amortized over the input domain can be represented as

$$\sum_{\vec{x} \in \mathbf{D}} \left[\tau(C_i,\vec{x}) - \tau(C_{mean},\vec{x}) \right]$$

If we refer to the inner term as Δ_i, we know that $prob(\Delta_i < 0) = P_i$. But knowing this value, and not knowing $|\Delta_i|$, we can not predict whether a performance increase will

[3] This statement ``know little'' must be qualified. On the one hand, the contents of a large personnel data base may be available before we begin our computation, so that we could "know" how long a particular formulation of a query will take. On the other hand, it seems computationally infeasible to (1) pre-compute this information or (2) store per-datum, or static aggregate, annotations of which method to choose. Also, execution time information from previous executions may have little predictive value, or may be sensitive to environmental change.

[4] For example, the single bit history of page-referencing behavior maintained with a ``dirty'' bit in a virtual memory management system.

result. With the advance of systems for process migration and load-balancing, determination and control of this value becomes more computationally difficult. The ``load-balancing'' can contribute to the observed execution times in such a way that it generates an unpredictable domain-wide dispersion [Barak1985a].

2.3. Conclusions

We have analyzed the properties necessary for a set of alternatives to show a performance improvement, in particular the relationship between the dispersion of execution times and the overhead associated with the parallel execution. These properties were general; for example, $\vec{x}$ may encapsulate the execution environment so that processors with heterogeneous performance can be accounted for.

The factors contributing to $\tau(overhead)$ were dissected; they are mainly a property of the execution environment and not of the application. Thus, measurements of *overhead* may allow us to calculate the performance improvement, *PI*, based solely on the application characteristics, thus removing a variable from any analysis.

3. Algorithms for Parallel Execution

3.1. System Model

A *process* is an independently schedulable stream of instructions. In implementations, it is often associated with some unit of state, e.g., an address space, and a set of operations provided by a *kernel* to manage that *state*. Interprocess communication is done solely through passing *messages*. Thus, a *message* is the only means by which:

- P_m can make P_j aware of a change in P_m's state.
- P_m can cause a change in P_j's state.

Interprocess communication (IPC) is assumed to behave reliably (no lost or duplicated messages) and FIFO (no out of order messages).

System *state* is divided into two types, *source* and *sink*. The division is made on the basis of idempotence; operations on *sink* devices can be retried without the effects being visible, while operations on *sources* cannot be retried. For definiteness, consider a page of backing store and a teletype device, respectively. Side effects which affect *sink* state can be hidden; this is a common technique in the implementation of such abstract operations as *transactions;* the idea is that the transaction has the property of *atomicity*, meaning that either none or all the transactions component actions occur, and that intermediate states are not observable external to the transaction. Complex transactions may involve reads, which can occur unhindered, or writes, which must be done to a temporary copy until the transaction *commits*, or in other words, makes its changes permanent. Reads intended for the recently written copy are satisfied by that copy so that the transaction is internally consistent, i.e., it can read what was written.

Sink state is manipulated as fixed-size *pages*. All sink state can be represented in this fashion; this is clear from implementations of a single-level store, as in MULTICS [Organick1972a]. Thus we bury the entire memory hierarchy under the page abstraction; files are named sets of pages, and thus mechanisms which are used to transparently access files over networks [Sandberg1985a, Weinberger1984a] can be used to hide the network through the page management abstraction. This has been successfully done in at least one commercial system, Apollo's Domain™ [Nelson1984a, Leach1983a, Leach1982a].

3.2. Process Management

Two primitives encapsulate the entire semantics of the process management component. The process management component is concerned with the mutually oblivious alternatives. To spawn the alternatives, the parent uses `alt_spawn( n )`, which returns numbers from 1 to n in the alternates and 0 to the parent. Thus a language preprocessor applied to a program with mutually exclusive alternatives would generate (in pseudo-C):

```
switch( alt_spawn( n ) )
{
   case 0:
      alt_wait( TIMEOUT );
      fail(); /* if returned */

   case 1:
      /* First alternate */
            .
            .


            .
            .
   case n:
      /* n-th alternate */
      alt_wait( 0 );
}
```

Figure 5: Use of alt_spawn() and alt_wait()

The purpose of `alt_wait()` is manifold: the essence is establishing a single path through the tree of possible computations which is reflected in the execution history of the running process.

`Alt_wait()` is the synchronization locus. `Alt_wait()` takes a TIMEOUT value as an argument: the point is that this value should be chosen such that if TIMEOUT time units have elapsed, it is highly probable that no alternative succeeded. While choosing such a value is hard, most computations have an execution time which is clearly unacceptable to the application: this value can then be used. The point of passing such a timeout value will be seen shortly.

When a spawned alternate calls `alt_wait()` at the termination of its computation, a rendezvous between the `alt_wait()`ing parent and the child is effected. The behavior is much like that of the UNIX `exec()` system call, where the new data and executable code are read in from a named file. For `alt_wait()`, the parent process absorbs the state changes made by its child by atomically replacing its page pointer(s) with that of the child. Thus, the flow of control through the child appears to have been seamless, up to and including maintenance of the process id. It is as if the parent process was "lucky," and performed the execution of the fastest alternative itself.

Use of these primitives is shown by concurrent execution of the program segment in figure 5 shown in figure 6:

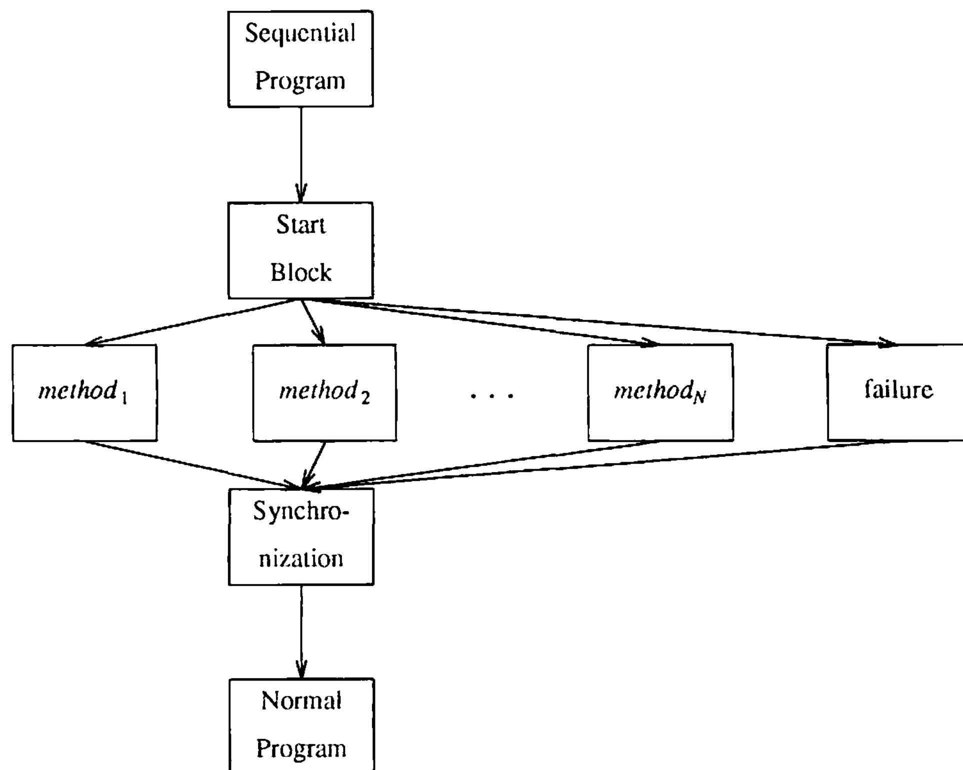


Figure 6: Concurrent Execution of Alternates

If all the GUARD conditions have been satisfied, a process which completes its program segment attempts to synchronize. If any of the conditions required by the GUARD were not satisfied, the process aborts without synchronizing. Note that the GUARD can be executed before spawning the alternative, in the child process, at the synchronization point, or at any combination of these places, for redundancy. We currently expect the

child process to execute it, thus speeding up spawning and synchronization.

3.2.1. Synchronization

It is at the synchronization point that the data for sibling elimination are available; all processes which assumed that the successful child had failed must be deleted, as they have made an assumption we now know to be false. To minimize the effect on throughput, when an alternative is selected, its ''siblings'' are eliminated. This is done by informing the scheduler that the process is to be terminated. The deletion can be done synchronously (where the other alternates are deleted before execution resumes in the parent) or asynchronously (where the deletion occurs at some time after the `alt_wait()` resumes in the parent, but exactly when is not specified); we suspect that asynchronous elimination will give better execution-time performance. This is because the execution time we are concerned with is that of the successful alternative. If the successful alternative cannot continue executing until its siblings are eliminated, then it is *waiting*, and thus will have increased execution time. Now, on the other hand, if the sibling elimination is started but the successful alternative does not wait for completion, it will result in faster execution, as work (and delay) has been removed from its execution trace. Measurements in Chapter 4 show that these suspicions are well-founded.

Now, communications problems or system failures may prevent this information from reaching the scheduling component of a remote system, yet we must still preserve the ''at most one'' semantics of our design. The backup in this case is that the synchronization action is designed so that it can be done at most once; that is, if the remote system attempts synchronization for the alternative it is executing, it is informed that it is ''too late'' for the synchronization, and it should terminate itself. In applications where this might create a single point of failure, the synchronization is set up as a majority consensus [Thomas1979a] decision across several nodes. The engineering tradeoff here is between performance and reliability; the additional communication and protocol of multiple-node synchronization is the price paid for increased robustness of the synchronization.

3.2.2. Atomicity

An important question is ``when do the alternate's changes become visible?.'' Clearly, this must occur at some point after the synchronization policy described above has been effected. Since we have taken the trouble to prevent the effects of other alternatives being visible, the update will be, by default, atomic (although we can have an ordered set of intermediate states made visible by the timestamping information we've preserved). Since any state changes must only be visible after the synchronization point, we'll assume that the state changes are made *atomically*, to simplify the discussion. It should be clear how the intermediate state changes can be made visible, but it seems pointless. How, then, can the state updates be made atomic? The method depends on cooperation between the method for supporting alternatives and the memory manager. Since we have required the predicates (described in the next section) to be stored in such a way that the *value* of a predicate is stored in one place, we can atomically update that information. Thus, an atomic state change is achieved as follows:

```
begin ATOMIC:
   All pages to be changed are predicated
      with FALSE;
   The predicating conditions (all TRUE) are
      removed from the alternate's pages.
end ATOMIC
```

Thus, it appears to other processes that all changes (to address spaces, or to what could normally be considered files) occur at once. The way in which I/O (e.g., printing a value) takes place must be considered carefully in an implementation, but the point at which the predicates can be removed from the pages is the same point at which I/O can be performed using these pages.

3.3. Predicates

Ideally, we would like an alternative to carry on with its computation as much as it can before either blocking or synchronizing. To effect this, we add ``predicates'' to the messages.

3.3.1. Representation of Predicates

Each page object has associated with it some set of predicates; these are the same predicates that are associated with message-sending. In most cases, there will be **NO** predicates associated with a page, as there is only one timeline, reality, that it is associated with; there are of course no conditions on reality.

Predicates might be stored as a bit-map, indexed by the process id, if the number of possible process ids is small (e.g., if it is limited by the Operating System's process table). If this is so, then predicate matches are essentially performed with existing machine instructions for comparisons, and rules for deciding whether a predicated object is accessible by a predicated process. Alternatively, the predicates can be stored as *lists;* in this case, there must be a more sophisticated, and possibly slower, comparison procedure. One idea is to store a linked list of process ids as a predicate; elements of this list can then be checked against their process table entries to determine the value of the predicate. If forward and back pointers[5] are used, we can update the value of these elements as processes change state, with the idea that processes change state much less frequently than they make memory references to objects. It seems both "fairer" and more practical to control state by using software-implemented per-process predicates. The arguments are as follows:

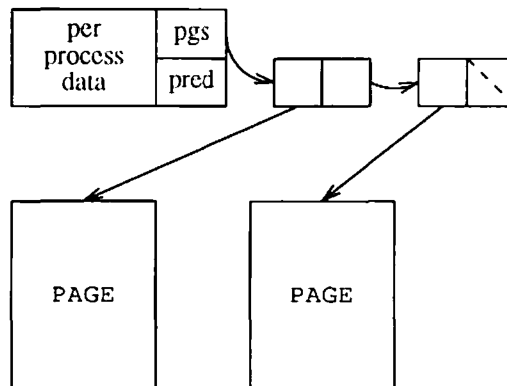
- Any mechanism which stores predicates at the page level must pervade the page management system. Thus, all users must pay the cost for the extra page management data, while not all users may have need for the data and page management facility.
- Predicates are used only in the process control and message transmission activities. Thus, predicate changes and evaluations are performed only at points where processes change state, not on a per-access basis.
- A bit-map for a system with 8192 processes would require 1024 bytes; thus for a system with 1024 bytes per page, there would be a page of predicates per page of data. Additionally, the comparison of predicates would have to take place at each page reference, thus slowing an already time-intensive operation. If the bit-maps are stored with the processes, the storage requirements grow as $O(n^2)$ for a bitmap, thus for 8192

[5] *Back pointer* here means that attached to the predicate is a list of processes which depend upon its value.

processes, 8 megabytes of bitmap storage would be required, and it is expected that most of it would be unused or sparsely used. There is a possibility for a time/space tradeoff since all bitmaps except for those of running processes can be stored in a compressed form. However, efficient compression algorithms often require significant execution time to compress and uncompress data, and this penalty would be paid at an already busy point, the context switch.

- With a list implementation, processes with no predicates would pay little (2 pointers) overhead, and testing this condition would require about two instructions: as has been pointed out in several performance studies [Leffler1984a, McKusick1985a], several thousand instructions are executed per I/O or context switch; thus the added overhead is significantly less than one per cent for processes which don't use the facility.
- As the system is described, there are three possible values which must be stored: "must complete," "can't complete," and "don't care." Thus, at least two bits per process are necessary, increasing the bitmap storage requirements by a factor of two. With lists, the presence of a process identifier in either the "can't complete" or "must complete" lists indicates that condition, and its non-presence in both lists indicates "don't care."

Thus, we will implement multiple "copy-on-write" [Bobrow1972a] forks to maximize sharing during parallel execution, and keep updated and newly-written pages linked in a per-process descriptor table. Pictorially,



One important notion is that the process-stored information must be dependencies, so that storage of the predicate values themselves can be logically centralized, even if this is

not the case in reality.

Two operations, READ_PAGE and WRITE_PAGE are defined: it's obvious that any value manipulation can be performed with these primitives. They provide the point of interaction between the process state, as defined by its predicate values, and the state of external values. These operations can be implemented as follows:

```
READ_PAGE:
   Obtain the PREDICATES of the calling PROCESS.
   Examine the PAGE TABLE for a PAGE
      such that No PREDICATES assumed FALSE
         by the PROCESS are TRUE.
   IF no such page is found, FAULT,
      setting the newly read page's
      PREDICATES to that of the PROCESS
   ELSE return a pointer to PAGE FI.
```

and

```
WRITE_PAGE:
   Obtain the PREDICATES of the calling PROCESS.
   Examine the PAGE TABLE for a PAGE
      such that No PREDICATES assumed FALSE
         by the PROCESS are TRUE.
   IF such a PAGE is found
      IF No PREDICATES assumed TRUE
         by the PROCESS are FALSE, update the
         PAGE and return a pointer to it.
      ELSE
         Make a new copy of PAGE.
      FI.
   ELSE
      Allocate a new PAGE.
   FI.

   Set the PAGE's PREDICATES to that of
      the PROCESS.
   Update the PAGE
   Return a pointer to the PAGE.
```

To see how this works, consider the following example, with Parent Process **P** having spawned Alternates $\mathbf{A}_1$ and $\mathbf{A}_2$. The predicate associated with $\mathbf{A}_1$ is $\mathbf{A}_1$ and $\neg\mathbf{A}_2$, which we'll represent as 10, and $\neg\mathbf{A}_1$ and $\mathbf{A}_2$, which we'll represent as 01. The state previous to the alternates is then 00, and 11 *can't happen*. The state 00 implies FAILURE after the alternates have been spawned, as it implies
$\neg\mathbf{A}_1$ and $\neg\mathbf{A}_2$. Let the initial state of the page map (**P**'*s*) be:

Virtual Page#	Predi- cates	Real Page#
0	00	19
1	00	12
3	00	3

Assume further that A_1 has the following operations: READ 0, READ 1, WRITE 3, READ 0, and that A_2 has READ 0, WRITE 0, WRITE 0, READ 3. The resulting state of the page map will be:

Virtual Page#	Predi- cates	Real Page#
0	00	19
1	00	12
3	00	3
0	01	37
3	10	8

The predicates are lists of process identifiers, some of which the sending process depends on completing successfully and others on which the sending process depends on *not* completing successfully. Thus, these are even simpler and easier to manage then the predicates described by Eswaran, *et al.*[Eswaran1976a] The advantage of this representation over predication of data objects is that we can update the value of these elements as processes change status (e.g., running, blocked), with the idea that processes change status much less frequently than they make memory references to objects. Note that since the processes are mutually exclusive, there is no problem sharing the other objects on a page.

These lists are constructed in two ways. First, the predicates of a ''child'' process consist of those of the ''parent:'' this allows for nesting and potentially complex dependencies. Second, when the ''parent'' spawns each of its alternative ''*children,*'' each of

the children additionally assumes that it will complete successfully, and that its siblings will not[6]. The state management strategy is ``copy-on-write'' [Bobrow1972a] with page map inheritance from the parent, thus it is easily implemented within the context of a system which provides such features, e.g., Mach [Young1987a], and benefits from existing hardware support, e.g., for the WE® 32101 Memory Management Unit [AT&T1986a]. The software-implemented predicates are used in the process control and message transmission activities to maximize sharing. Updated and newly-written pages are predicated by virtue of their residence in a per-process descriptor table.

3.4. Interprocess Communication

3.4.1. Messages

A *message* from P_m to P_j has the following three part structure:

1) A sending predicate, encapsulating the assumptions under which the *sender,* P_m sends the message.

2) The data comprising the message contents.

3) Some control information, e.g., sender id, destination id, etc.

Each *process* in a *multiprocessing* (e.g., timesharing, multiprocessor, or distributed) system has a *unique identifier*. used to identify the process both within the system (e.g., for scheduling and resource allocation), and further, for interaction with other processes. This *unique identifier* can be constructed by concatenating several quantities at the time of process creation if local process ids are not unique (as in a distributed system), but processor names are:

```
<processor name, local process id, timestamp>
```

The timestamp is included to mitigate against re-use of local process ids. In the single-processor case, a local process id must be unique at any given time in order for the process to be named in an unambiguous fashion. However, as soon as the process terminates, the name, e.g., a table address or small integer, can be re-used. The local

[6] Thus, so-called "sibling rivalry" is taken to its extreme in this design! The failure alternative assumes that none of the siblings will complete.

picture of time may not hold across processors, and thus we force uniqueness with a timestamp.

3.4.2. Multiple Worlds

An idea from science fiction, inspired by Dewitt's [DeWitt1973a] multiple worlds notion, is appropriate here. The problem with interprocess communication stems from the fact that a given alternative may or may not be successful. In the case where it is successful, its execution results are available to the calling process. Where it is not successful, its results and any side-effects it may have generated must not be observable. These include side-effects due to interprocess communication.

The specialized side effects resulting from interprocess communication are controlled by a message layer which insures that any receiver of a message makes the same assumptions about the state of the world as the sender. Since a given sender may fail and make its assumptions invalid (as well as irrelevant) two copies of the receiver are created when the sent message causes the receiver's state to change: one copy is in the sender's "world," the other is not.

In any case, the message subsystem must be aware of the decisions made, so that the copies are made *once*, at the first message receipt, not at each message receipt. Thus, as far as the receiving process is concerned, there are two worlds: one where the sender exists, and one where it doesn't. Depending upon whether the sending process completes successfully, one "world" may be realized.

The message system, the virtual addressing mechanism, and the process management mechanism are linked in the following way. When a receiving process accepts a message, its predicates (**R**) are checked against those attached to the message (**S**). If the assumptions that the receiver makes about the "state of the world," as encapsulated in the predicates, agree with those of the sender (e.g., $S \subseteq R$), the message is immediately accepted. If the receiver's predicates conflict ($p \in S$ and $\neg p \in R$), the message is ignored, and if the receiver must make further assumptions to accept the message ($p \in S$ and $p \notin R$), two copies of the receiver are created. One copy is created with the predicates set to the previous values with `complete( S )`[7]: the other is set up with its predicates as

[7] Thus implying all the sender's predicates.

before, except that `complete( S )` is negated.[8] This is shown in a revision of a previous figure:

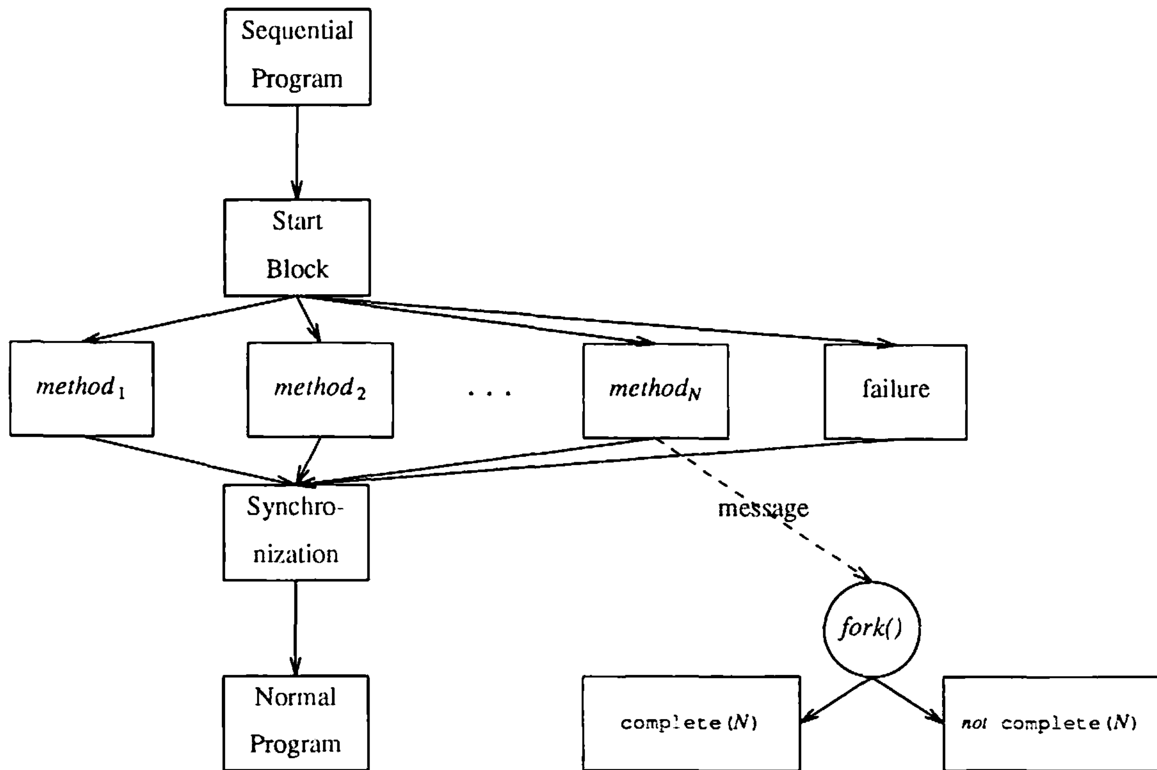


Figure 7: Use of predicates

This is easy given the representation as two lists (i.e., "must complete" and "can't complete") of process identifiers. When the sending process succeeds or fails, one receiver must be eliminated to maintain a consistent "state of the world;" at this point the additional assumptions which receipt of the message caused will become `TRUE`, and they can be eliminated from the lists.

To illustrate the idea, consider a group of communicating processes composed of P, Q, and S. P has children a, b, and c; Q has children d, e, f. This is illustrated by figure 8.

[8] Thus implying rejection of the sender's predicates without creating a logical impossibility. Assuming the negation of *all* of S's predicates might imply that two mutually exclusive processes must complete.

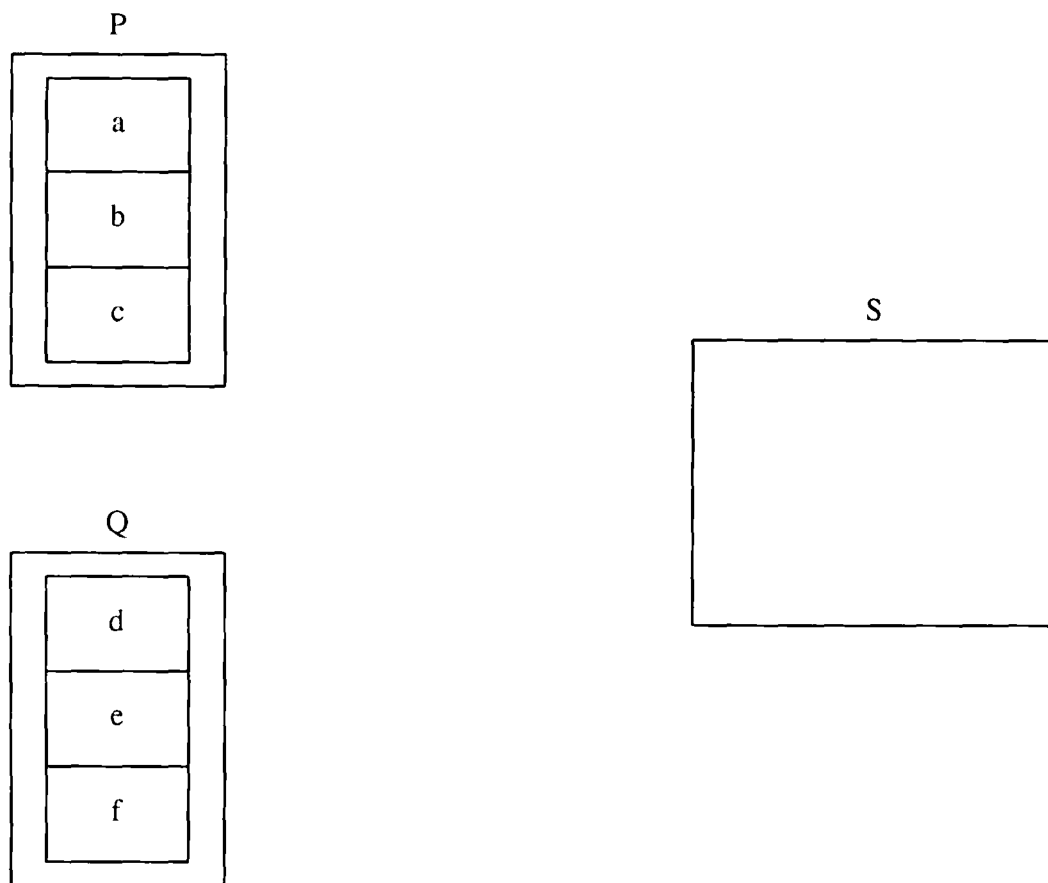


Figure 8: Communicating processes

Suppose both b and f wish to communicate with S. We use the predicates to create multiple copies of S, with which the alternatives communicate. This is illustrated in figure 9.

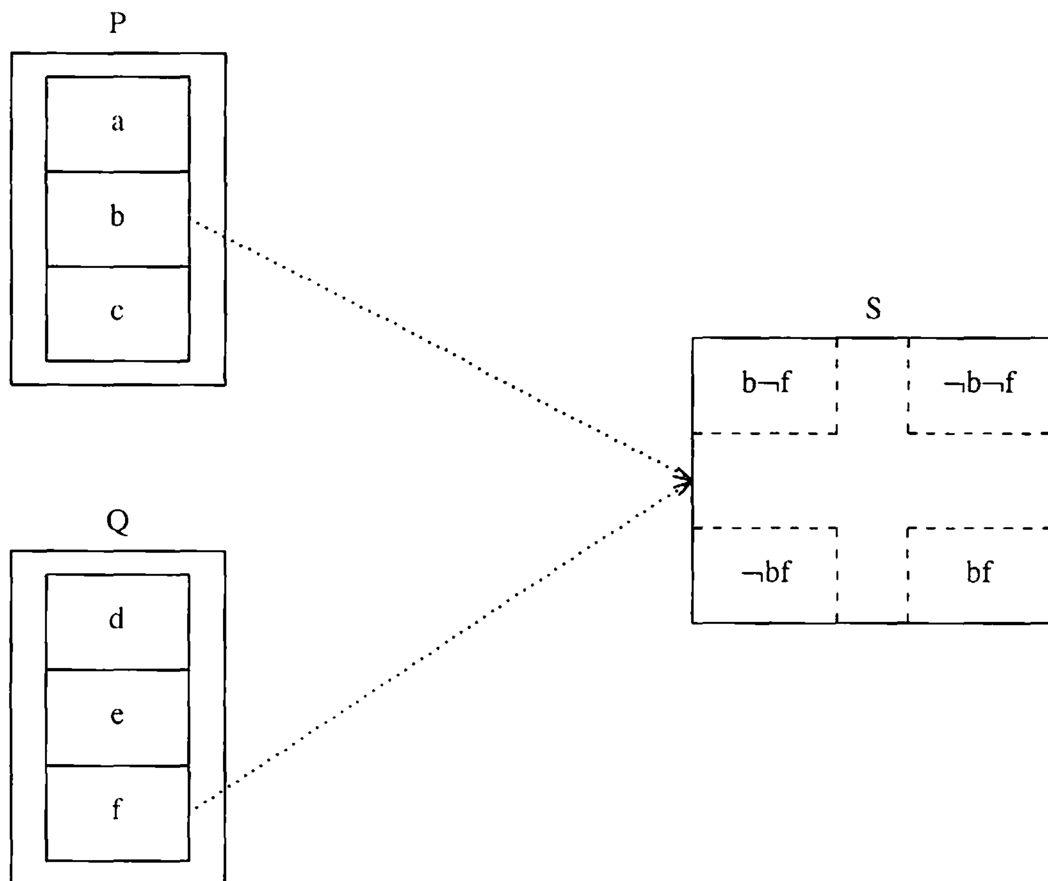


Figure 9: Communicating processes after message receipt

There are several implications to this scheme:

1. There is clearly a potential for a combinatorial explosion in the number of processes that exist, and amount of state which must be copied. Since the process creation is based on message receipt, it is "lazy".

2. Commit is *very* fast, as it can be accomplished using only pointer (page descriptor) manipulation. For example, ¬bf can be committed by simply updating the page descriptor table associated with S.

3. Messages must be duplicated, as all valid receivers must get the message.

4. The method is *optimistic*. Rather than locking a resource based on a predicate, the assumption embodied in the predicate is sent with messages and used to create new "worlds."

While a process has predicates which are unsatisfied, it is restricted from causing

observable side-effects, and thus cannot interface with *sources*.

This behavior is similar to that required of *transactions*. Transactions [Gray1978a, Lampson1981a, Traiger1982a, Gray1981a] are a structuring concept for operations: transactions are required to be atomic with respect to any observer. This atomicity property means that a transaction either executes in its entirety or does not execute. Thus, any side-effects of a transaction which is in progress (i.e., not complete) must not be visible, since the transaction might fail. The method described here might be viewed as a set of competing transactions, at most one of which will complete, or *commit*. The competing transactions must not only be isolated from external computations, but they must be isolated from each other as well.

3.5. Discussion

Upon receipt of a message, the predicates associated with a message (by virtue of it being sent by some process with assumptions about its existence and the existence of other processes) are checked against those of the receiver. Remember, the predicates associated with a process encapsulate the assumptions it has made about the world in order to continue executing. It has to be done this way to avoid *blocking* or *waiting* for the predicated condition (typically successful termination) to become true. The idea is that for some process these assumptions will be right. In order that the interaction with other processes leave a consistent "world view," we are forced to perform some special actions upon message transmission between two processes, e.g., one alternative and some server process. The idea is that the message layer, as illustrated above, is involved in process management as well. When a process receives a message which causes it to update its state (read-only messages don't change the receiver's world view, and so they can be ignored by our mechanisms but are of course used by the receiver), it must be careful. The care is taken in examining the assumptions which must be made by the receiver to continue executing. If a sending process has made an assumption which the receiver has not made, the receiver has two choices. The receiver can share the assumption with the sender, and reply accordingly. Or, the receiver can reject the assumption, and hence the message and implied updates of its state. If the sending process completes successfully, then its updates become visible when the predicates are removed from its state and from the receivers, but not before.

One issue which should be discussed is *locks*. If one of the alternatives successfully obtains a lock on a data object, then other alternatives will be blocked and the concurrent execution will not be transparent. Two scenarios are possible where a lock is involved. If the lock is acquired before the alternative block is entered, a copy of the state indicating successful lock acquisition will be made when the alternatives are spawned. Copies of the locked object will be made as the object is changed, and one of the set of changes will be available after the synchronization point is reached. The other scenario is lock acquisition by an alternative. In order for our mechanism to work in preserving the transparency, the lock must be accessed through another process. Then, the predicated message mechanism can be used in lock acquisition, so that two ``worlds'' exist, one where the lock has been acquired, and one where it has not. Otherwise, there are a variety of situations from which inconsistency or deadlock could arise.

The opportunity for a performance increase arises when the methods require different amounts of execution time.

This is illustrated in the timeline diagram of Figure 10:

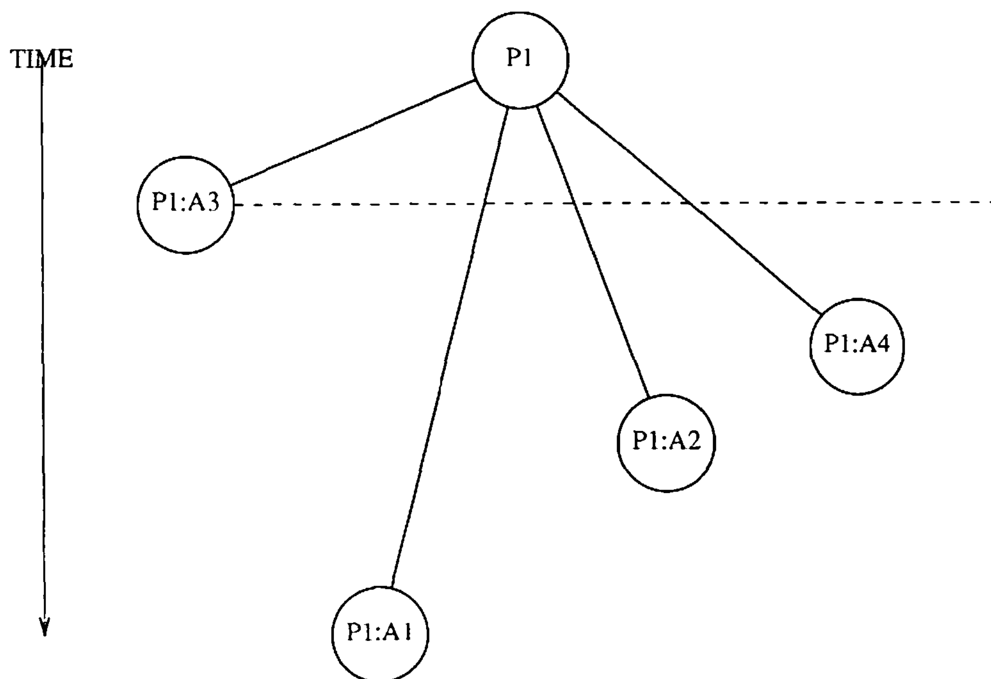


Figure 10: Alternates vs. Time

It's clear that Alternate A3 is the fastest. If this had been predictable *a priori*, the

programmer probably wouldn't have used alternatives. We'd like to take advantage of the fact that A3 executes most quickly by selecting it and deleting the other alternatives; this is done by ''sibling elimination.'' This is done because any computation done after A3 has terminated (the point in time indicated by the dotted line) is wasted computation.

In the next chapter, we evaluate implementation strategies, and the *overhead* associated with several execution environments. The careful performance measurement methodology is of particular interest, since this need be done only **once** for a particular execution environment. Several application areas are discussed, and experimental results demonstrate that speedup is possible with the competitive scheme on problems where no cooperative solution is possible.

4. Implementation, Applications and Experiments

''*...the sole test of the validity of any idea is experiment.*'' [Feynman1963a]

4.1. Measurement of Overhead Costs

It is informative to examine measured values of possible contributors to τ(*overhead*). Earlier, we described these as the cost of creating alternates, the cost of maintaining the alternates, and the cost of sibling elimination.

We have developed analysis techniques and software to evaluate the sources of overhead. The analysis techniques were applied on two workstations, the AT&T 3B2/310™ and the Hewlett-Packard HP9000/350™

We begin by determining the relationships between the amount of memory in the parent's data segment, the fraction of this memory which is written by the child, and the improvement in execution time due to ''copy-on-write.'' Since the implementation of ''copy-on-write'' is straightforward with modern [AT&T1986a] memory management units (MMUs), our results for these workstations are readily generalized to other workstations.

The results show that the size of the parent's allocated memory has little direct effect on performance, because only page table entries are copied during the *fork()* operations. The execution time is most influenced by the amount of memory that must be copied, which can be determined from the product of memory allocated and the fraction of memory written. Thus, the worst case occurs when large address space programs update much of their memory.

To observe what occurs in practice, we measured two programs that have what are currently considered large address spaces. These programs, which we believe to be representative of the sorts of programs which use large amounts of system resources, updated less than half of the memory in their data segments.

4.2. Copy-on-write

The UNIX® *fork()* operation creates a copy of the calling process which is differentiated from its creator by the return value of *fork()*. The two processes have separate address spaces. Traditionally, UNIX systems copied the contents of the caller's address space to create the new process. Since the portion of the address space containing executable code was read-only, copying was not needed and an incremented reference count and text table entry sufficed [Ritchie1978a]. Clearly, the *fork()* operation can be expensive in system resources. Thus, some attempts were made to take advantage of special cases. An example is the 4.2BSD [Joy1982a] *vfork()* call, which does not make a copy of the address space for the new process but instead allows it to share the address space with its creator. The creator is not runnable until the new process has replaced its image via an *exec()* operation. The *exec()* operation replaces the caller's image with an image derived from the contents of the named executable file. It is common for the operation which immediately follows a *fork()* operation (after some descriptor manipulation) to be an *exec()* operation. In particular, this happens frequently in the shell [Bourne1978a], which is the main user interface to UNIX. Thus, *vfork()*, in not copying, avoids unneeded work. However, the shared, not copied, address spaces force the programmer to be very aware of the differences between *fork()* and *vfork()*.

Another approach is to alter the implementation of *fork()* to take advantage of favorable circumstances such as the shell's usage. This change should be made *transparent* to the application. The alteration is done with a so-called "copy-on-write" *fork()*, where portions of addressable memory are shared until they are changed. Similar memory management is done in TENEX [Bobrow1972a] and more recently, Mach [Young1987a]. Each process has a page table which maps its virtual addresses to physical addresses: when the *fork()* operation is performed, the new process has a new page table created in which each entry is marked with a "copy-on-write" flag; this is also done for the caller's address space. When the contents of memory are to be updated, the flag is checked. If it is set, a new page is allocated, the data from the old page copied, the update is made on the new page, and the "copy-on-write" flag is cleared for the new page. Thus, unexpected changes to shared state do not occur, as independent copies are created "on demand." This is effective in the special case of the shell, where almost no copying has to be done before an *exec()* replaces the address space. A thorough

description of the mechanism as implemented in UNIX is given by Bach [Bach1986a].

4.2.1. Motivation

In section 4.3, we present results from a paper [Smith1989a] where we discuss an implementation of a mechanism to *fork()* a process on a remote workstation; the major cost in execution time is incurred by data copying. Thus, we were interested in reducing the amount of copying, especially that which takes place over a communications channel. One strategy which we devised (assuming either homogeneous software configurations on the workstations or NFS-available [Sandberg1985a] binaries) was to have program images available on the remote system and send only the changes [Maguire,Jr.1988a] which have been made to the address space, i.e., those which would be copied by a ''copy-on-write'' scheme. To understand the engineering tradeoffs, we examined the local case in some detail.

The arguments presented for ''copy-on-write'' have so far been qualitative; we felt that detailed quantitative data were necessary.

4.2.2. Data Acquisition

There are two parameters of interest, i.e., the size of the storage to be ''copied'' in the new process and the fraction (between 0.0 and 1.0) of memory references which are writes. The number of times each parameter was exercised was also made variable, to remove various small-sample artifacts that can occur. Such artifacts are illustrated by the plots for small sample sizes in the copy-on-write *fork()* measurements. The desired data were gathered with the C program presented as Appendix I, *do_fork.c*. A script was written in order to drive the *do_fork()* program with various values: the values used for the measurements described here were gathered with this shell script:

```
if [ ! -f do_fork ]
then
   echo "Making do_fork."
   make do_fork
fi
if [ ! -f do_fork ]
then
   echo "No do_fork. Exiting."
   exit 1
fi
echo "size do_fork:"
size do_fork
for forks in 0 1 3 10 32 100 316 1000
do
   for heap_size in 0 1000 3162 10000 31622 100000 316228
   do
      for write_frac in 0.0 0.1 0.3 0.5 0.7 0.9 1.0
      do
         echo "time do_fork $forks $heap_size $write_frac"
         time do_fork $forks $heap_size $write_frac
      done
   done
done
```

The script first ensures that an executable *do_fork* binary is available, attempting to make one if not. Once *do_fork* is available, it is invoked in the innermost of three nested loops, which vary its parameters controlling the number of *fork()* operations to be executed, the size of the heap to allocate, and the fraction of the allocated heap which is to be written to. Before each invocation, a message is written with *echo,* stating what the invocation parameters of *do_fork* are.

Data sets for analysis by S [Becker1984a] are then created using the shell script, by, e.g. for the 3B2,

```
script 2>&1 |\
   grep "^real" |\
   cut -f2 |\
   awk '{ i=index($0,m); m=substr($0,1,i-1);\
   s=substr($0,i+1,length($0)-i-1); s=60*m+s;\
   print s}' > real.3B2
```

and reading the list of numbers into an S vector. The following data sets were extracted from the script output:

number: The number of times an invocation of *do_fork* was to create a child process. The values 0, 1, 3, 10, 32, 100, 316 and 1000 (0 plus powers of *sqrt(10)* were selected to make both order of magnitude induced effects (as we are changing by orders of magnitude) and implementation artifacts (because we start at small values, e.g., 0 and 1) visible.

mem: The number of bytes allocated to the process's heap, via *malloc()*. The values 0; 1,000; 3,162; 10,000; 31,622; 100,000 and 316,228 were chosen for both artifact and order of magnitude visibility, as discussed previously: the extra factor of 1000 (over the values of **number**) is to compensate for the page size, since otherwise it would require (for a 2K page[9]) 8 values before we accessed a page other than the first one. Clearly, there is no practical difference between 316,228 and 310K: it is merely aesthetically appealing to use the correct digits.

frac: The fraction of memory which is to be written (we write one byte per page in order that the memory access loop not contribute to the response time beyond causing faults). The interesting boundary values of 0.0 and 1.0 were chosen, as well as the values 0.1, 0.3, 0.5, 0.7, and 0.9, which were chosen for their coverage of the input domain.

real: The real time, in seconds, printed by an invocation of ``time do_fork'' with the parameters as set in the other vectors.

user: Likewise for user time.

sys: Likewise for system time.

4.2.3. Data Analysis

Given the data discussed in the previous section, we wish to analyze the data in order that we can qualitatively discuss the effects of ``copy-on-write'' page management on response time. One difficulty is that by our experimental design, the measured response time is a function of not one, but three quantities, **number, mem,** and **frac.** There are

[9] HP-UX on our HP9000/350 systems uses a 4K pagesize. With our instrumentation (e.g., do_fork.c) the difference is relevant only when the offset of a particular byte in the last page accessed causes an extra 2K bytes of memory to be paged in. It may be more relevant to applications.

two obvious hypotheses which we can propose for our analysis to refute or verify. First, that the response time increases as the size of the data segment increases, for a fixed fraction of write references. Second, that the response time increases as the fraction of write references increases, for a fixed data segment size. Figure 11 shows

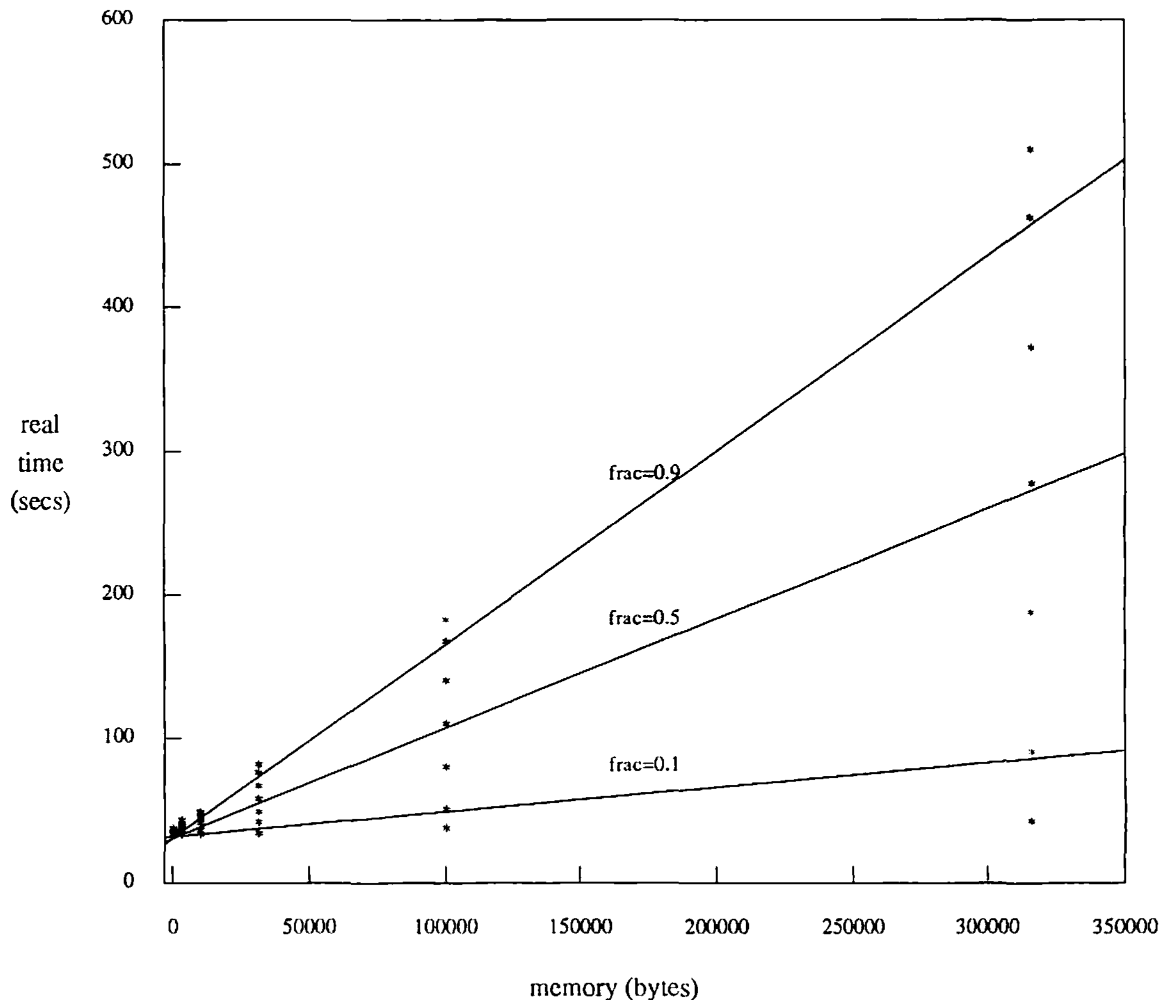


Figure 11: Effect of fraction of memory written

(number = 1000)

mem plotted on the x axis against **real** on the y axis for an AT&T 3B2/310 with 2 megabytes of memory (of which 1.2 megabytes are available to user processes), a 30 megabyte hard disk, and running UNIX System V, Release 3.0, Version 2. All times are given in units of seconds. We have fixed the value of **number** to be 1000 to remove artifacts. The dependent variable, plotted vertically, is the real time, in seconds. The independent

variable, the size of the data memory in bytes, is on the horizontal axis. Regression lines are drawn through the plotted points corresponding to **frac** values of 0.1, 0.5, and 0.9. These regression lines have equations $y = 1.709e{-4} \cdot x + 31.4$, $y = 7.670e{-4} \cdot x + 30.5$, and $y = 1.349e{-3} \cdot x + 30.7$ for the respective **frac** values. Thus, with these equations, we could estimate that a process with a 1 megabyte data segment which writes into half of that segment would take about 800 ($797.5 = 7.67e{-4} \cdot 1.0e6 + 30.5$) seconds of real time to perform 1000 *fork()* operations. The lines fit the plotted points well, indicating that the relationship is close to linear.

The same data are plotted for a Hewlett-Packard HP9000/350 with 8 megabytes of main memory and an HP7945 70 megabyte hard disk, running HP-UX™ 6.0 (same units, restrictions, and axis markings) in Figure 12.

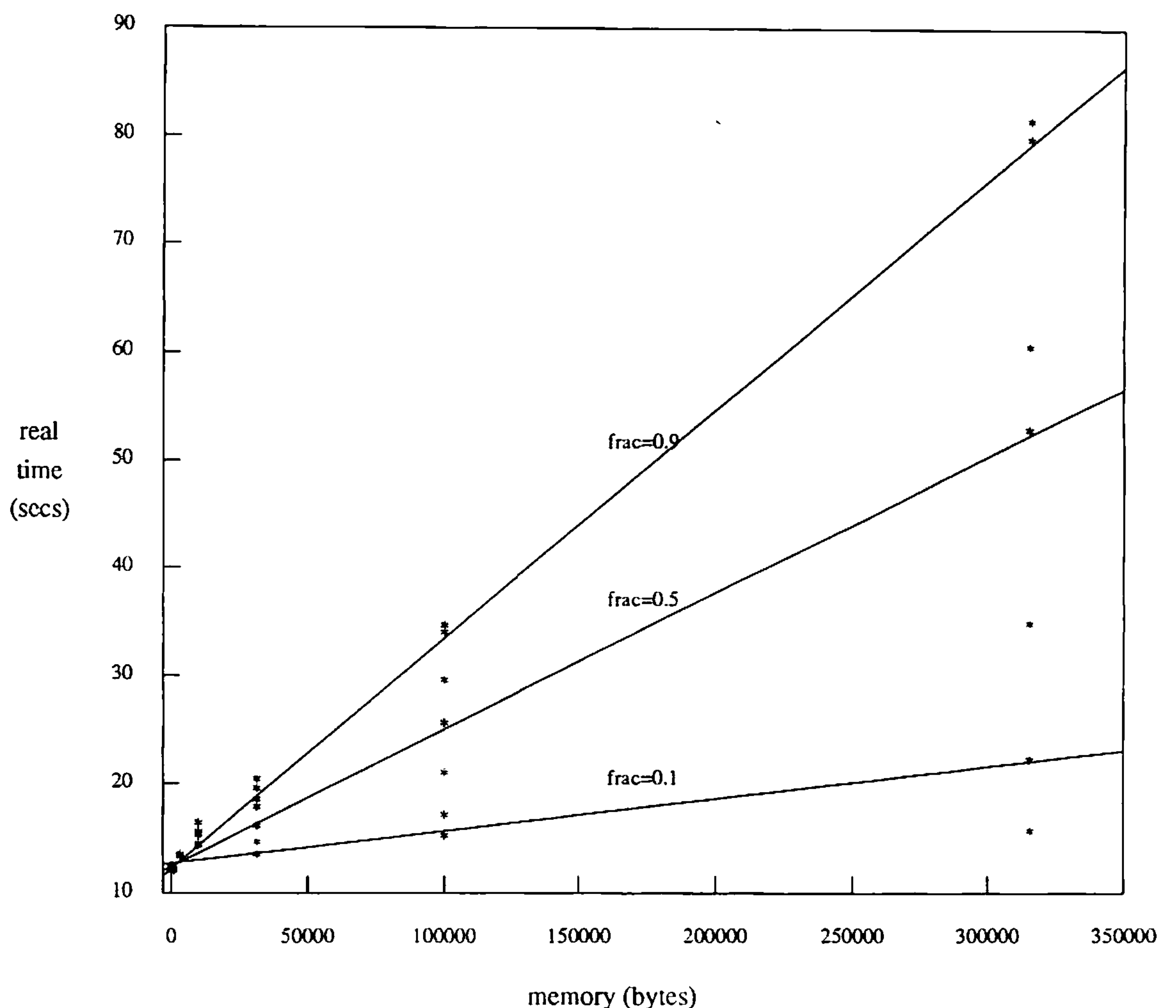


Figure 12: Effect of write fraction (HP-UX)
(number = 1000)

The equations for the lines with **frac** set to 0.1, 0.5, and 0.9 are y=2.952e−5·x+12.7, y=1.264e−4·x+12.4, and y=2.124e−4·x+12.2, respectively. The effect of the faster processor in the HP9000/350 is clear from the extent of the y axis in this figure versus that of the previous one. The important parameter in comparing processor speeds under this workload is memory-copying speed. To measure this, we wrote a short C program which took the number of bytes to copy as an argument, the relevant fragment of which is:

```
p = malloc( size );
pre_page( p );
clock = times( &tbl );
memcpy( p, p, size );
clock = times( &tb2 ) - clock;
```

For **size** set to 316,228 and a page size of 2K bytes (4K on the HP9000) we measured 0.40 seconds of real time, 0.39 seconds of user time, and 0.00 seconds of system time on the 3B2/310. The values were 0.06 seconds of real time, 0.06 seconds of user time, and 0.00 seconds of system time on the HP9000/350. These values held true through several trials, and show that for memory-copying the HP is about (to the limited accuracy of the measurements) 6.7 times faster than the 3B2. They also provide an upper bound on the memory copy rate which can be used to evaluate overhead incurred by page management operations. For the HP, we get 5M (5,270,467=316,228/0.06) bytes per second, or about 1,300 (1,286=5,270,467/4,096) 4K pages[10] per second. For the 3B2, we get 0.8M (810,841=316,228/0.39) bytes per second or about 400 (396=810,841/2,048) 2K pages per second.

We can use the regression lines we have presented for further analysis. The y-intercept (about 31 seconds for the 3B2/310 and 12 seconds for the HP9000/350) should represent the time required for 1000 forks which allocate 0 bytes of memory: examination of the script output confirms that this figure is accurate. Since *do_fork* is written to be compact (no standard I/O, etc.) this should accurately indicate the cost of performing a *fork* when divided by the number of operations performed. Thus, using the computed intercepts we have given for **number** set to 1000, the average 3B2/310 *fork* requires about 31 (=(31.4+30.5+30.7)/(3·1000)) milliseconds of real time. For a fixed number of *fork()* operations the y-intercept is not nearly as interesting as the slope of the line. We should note that in reality, the function is not a line, as the quantization of bytes into page size quantities forces a staircase function. However, for purposes of analysis we can assume that a linear function exists. The slope of the line for some known value of **frac** gives the relationship between changes in **real** caused by changes in **mem.** Hence, we can use the slope of the regression line to estimate the rate at which page faults are

[10] For comparison purposes, this would be about 2,600 (2,573=5,270,467/2,048) 2K pages per second.

serviced. **Mem·frac** gives a fixed amount of memory, with which we use the equation of the regression line to compute a real time estimate. Then, the observed page fault service rate can be computed with the simple formula $\frac{\textbf{mem}\cdot\textbf{frac}}{\textit{real time}}$. The slope of the line can be used to compute the service rate directly, for a known value of **frac** and **number;** this rate is given by $\frac{\textbf{number}\cdot\textbf{frac}}{\textit{slope}}$, which calculates a value in units of bytes per second. For the 3B2, these values are 585,138; 651,890; and 667,161 (286, 319, and 326 2K pages/second, respectively) for the three values of **frac** plotted. The corresponding values for the HP9000 are 3,387,534; 3,955,696; and 4,237,288 bytes per second (827, 965, and 1,034 4K pages/second[11], respectively). Using the best observed page fault service rates for each processor, we calculate the ratio of the page fault service rate and the time for memory-copying, which is 0.823 (=667,161/810,841) for the 3B2, and 0.804 (=4,237,288/5,270,466) for the HP. Values for the ratio can range between 0 and 1; the best case is a value near 1, as this indicates that the virtual memory management incurs little overhead. We can estimate this overhead using the information we have. Using $\tau()$ to measure time, we know that

$$\tau(\textit{fork}) = \tau(\textit{copy one page})\cdot\textbf{frac}\cdot\textit{\# pages} +$$
$$\tau(\textit{overhead for page table entry})\cdot\textit{\# pages} +$$
$$\tau(\textit{overhead to create new process})$$

Now, τ(*copy one page*) is really a function of the hardware components (e.g., bus, processor, memory) comprising a system, and we've shown how it can be gathered with a small auxiliary program. But from our numbers and analysis we can get τ(*overhead for page table entry*) and τ(*overhead to create new process*). Thus, for any given *fork* operation, the time required is completely parameterized by τ(*copy one page*), τ(*overhead for page table entry*), τ(*overhead to create new process*), **frac,** and **mem** (# pages). The key is that the first three are determined by the system characteristics, and they can thus be precomputed; the application-dependent influences are completely encapsulated in the latter two parameters. Thus, an application can be characterized on a given system by its **size in pages** and the **fraction of those pages which are written to.**

[11] For comparison, 1,655; 1,932; and 2,069 2K pages/second.

4.2.4. Relationships

The shapes of the plots we generated are similar for both processors. The HP9000 plots' time values (the y axis) are scaled differently because the HP9000 is significantly faster than the 3B2. We'll use the 3B2 to illustrate the analysis in the remaining figures.

One failing of the x-y plots is that there are two independent variables. The perspective plot shows that the real time increases as a *product* of **mem** and **frac**; the maximum value is 505.82 seconds, for **number** at 1,000; **mem** at 316,228; and **frac** at 1.0. Figure 13 shows a 3-D perspective plot of **real** (z-axis), **mem** (x-axis), and **frac** (y-axis). The point (316,228;1.0;505.82) is the furthest, highest point on the graph. Thus **mem** is increasing from our left to our right (it's an exponential curve as the data values are chosen to increase exponentially), and **frac** is increasing from our right to our left, moving away from us. Figure 13 would then be overlaid cross-sections taken from the perspective plot by intersecting a series of y-planes with it. Of course, not all the data of figure 13 are available due to hidden line elimination.

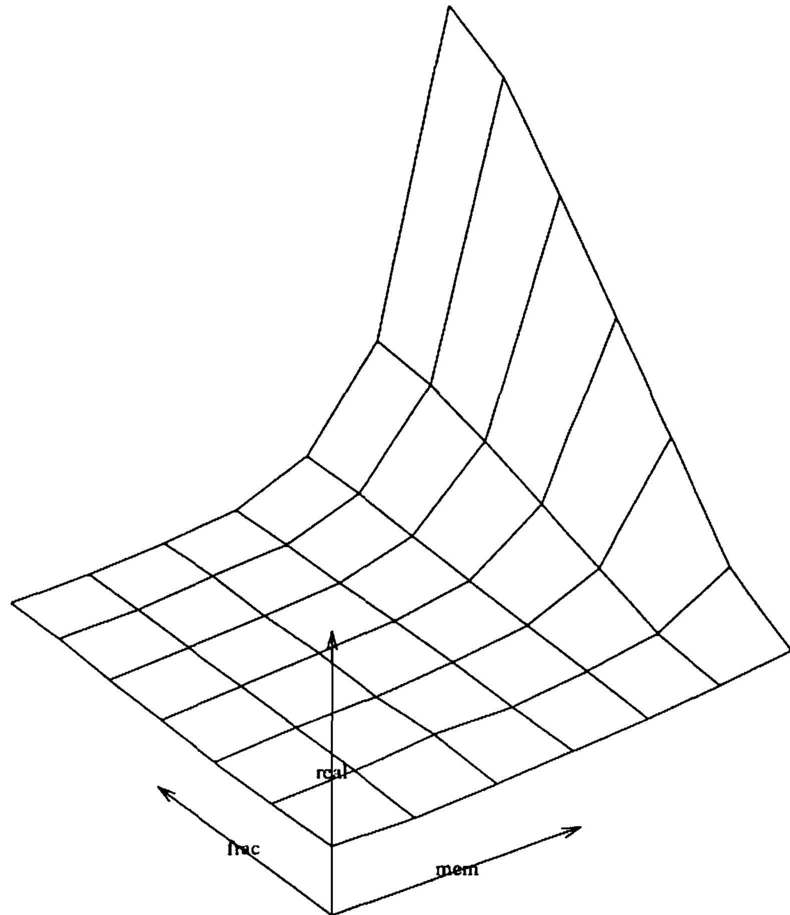


Figure 13: Perspective plot, mem vs. frac vs. real

We've limited the data shown in Figure 13 to that gathered with **number** set to 1000. This was done after analysis of the raw data showed two things which limited the value of the data gathered for a small number of *fork* operations. First, there was little opportunity for the data to become evident against the overhead of executing the parent program. This could, of course, have been removed by calling *times()* from inside *do_fork*, but given our strong preference for the shell as a measurement apparatus, this was not done. Second, the timing data were apparently overwhelmed by other sampling noise, such as that caused by various background processes and network daemons (although the processors used for these tests were otherwise idle). These other processes were not shut down due to the effect on our working environment.

If we plot a 3-D perspective plot with parameters as before, except that **number** is 1, we get Figure 14, which demonstrates what sort of artifacts, or ''noise'' can arise due to inadequate sample size.

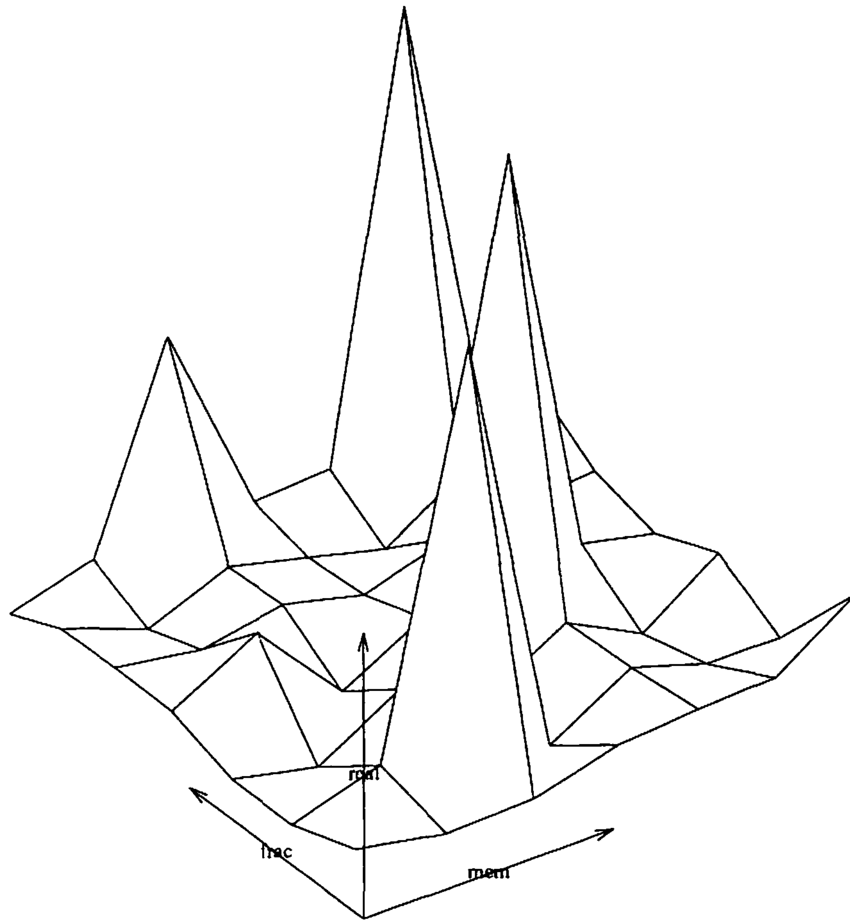


Figure 14: Perspective plot, mem vs. frac vs. real

The deduction one can make from examining this plot is that there is no obvious relationship between the input values and the response time output. The errors and spurious values have dominated the measurements to the extent that visual tools such as graphs are no longer useful. It is doubtful whether any tools are useful under these circumstances, and the lesson is clear: the analysis must be aware of the sources of error, and the measurements must be made in such a way as to minimize these sources. In our case, the minimization was achieved by using an adequate sample size.

The question might be raised as to why **real** time is used, not **sys**. Philosophically, the **real** time is what is most relevant to an observer. Scientifically, analysis shows that for **number** large, **real** is less than 20 percent greater than **sys**, and that they are closely correlated. This is illustrated in figure 15, where the x axis has values of **number**, and the y axis is the value $\frac{\text{real}-\text{sys}}{\text{sys}}$.

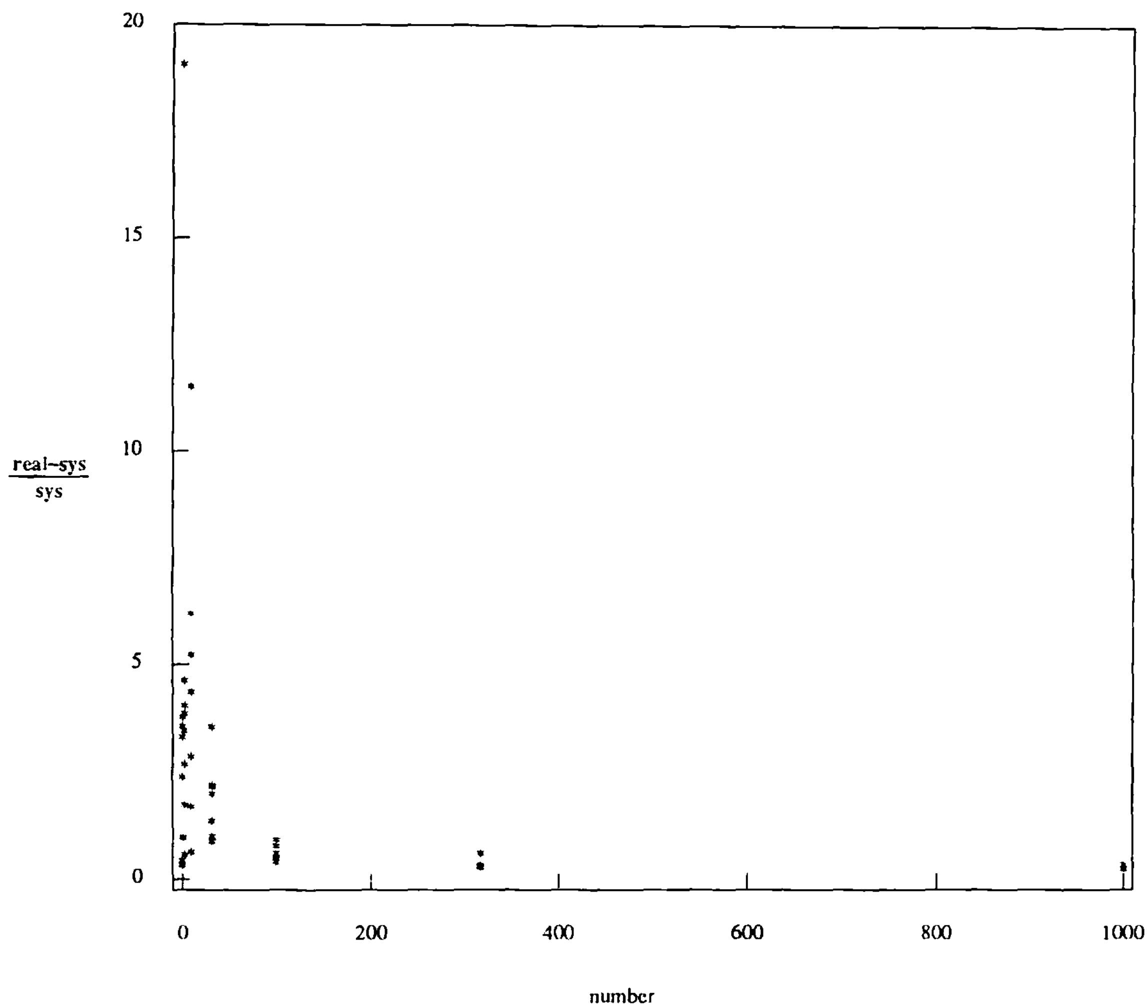


Figure 15: Relation between **real** and **sys**

The plot shows that the relationship between **real** and **sys** is not good for small values of **number**; they differ by almost a factor of 20. However, things improve as **number** gets larger; a detailed graph is provided in figure 16 by restricting **number** to values of 100 or more. It's clear from this illustration that for **number** at 1000, sys and **real** are

reasonably good approximations of each other.

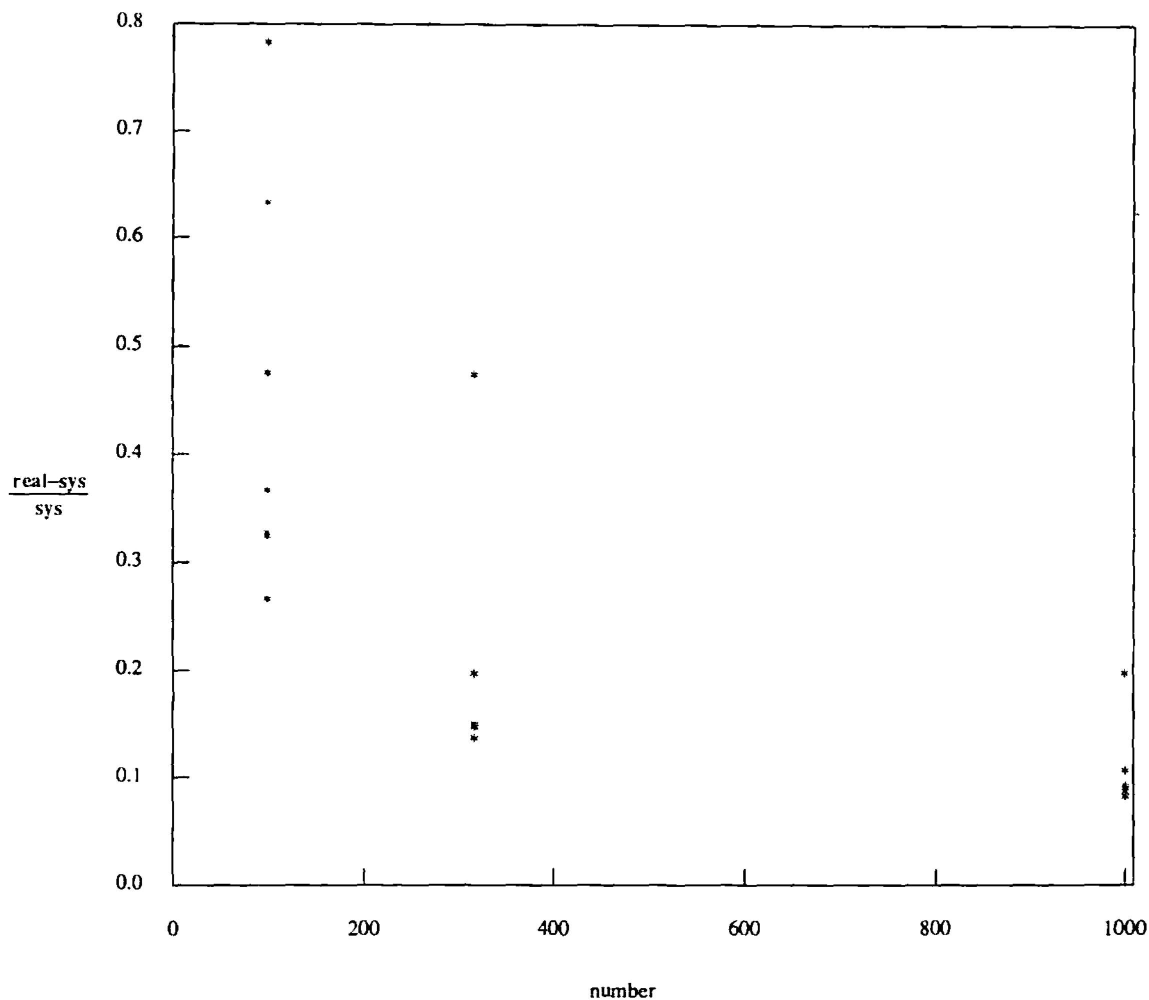


Figure 16: Relation between **real** and **sys**

Incidentally, we should note that for small values of **number** (e.g., 1), **sys** is subject to the same noise problem that **real** suffers from; this is easily observed with another perspective plot, which we will not present due to space considerations.

4.2.5. Write Fraction for Real Programs

In the last section, we saw that the factors **mem** and **frac** influenced the real time requirements of our test program. The biggest savings for the "copy-on-write" scheme would come from programs with large address spaces which updated a small fraction of their

data before exiting or *exec()*ing a new binary. As discussed before, this works well for the shell, but the shell typically uses little of its data segment. While it may expand the address space as necessary to store new variables or metacharacter expansions, this does not account for many pages. We thus sought programs with large address spaces, to see what effect the ''copy-on-write'' scheme would have.

As they had the largest address spaces (of programs in common use in our department), we set out to take some measurements of the memory utilization of two symbolic interpreters. We chose 4.2BSD's [BSD1982a] Franz Lisp (Opus 38.92), as it is widely available. Another less detailed set of measurements was taken using the GNU Emacs [Stallman1986a] LISP interpreter, which is also widely available. These measurements were taken on a DEC™ VAX-11/750, because both pieces of software were available there (Franz Lisp is not available on our HPs and 3B2s, although GNU Emacs is). Since we are measuring data segment utilization, and the machines discussed in this paper all have 32 bit architectures, the measurement results should be portable. This is particularly true because we use relative measures, such as the fraction of the data segment which has changed. While a particular architecture may have a less efficient representation of the data, this should not change the fraction of the data altered by the program significantly.

4.2.5.1. Franz Lisp

Our first exercise was choosing a computationally intensive process so that we could gather some statistics on the sort of processes which one would want to improve the performance of [Leland1986a] ; that is, those that consume many resources. Experience with an ABSTRIPS [Sacerdoti1974a] implementation led us to use this system to gather statistics. ABSTRIPS is a ''planning'' system which works by constructing increasingly detailed series of actions at decreasing levels (''criticality levels'') of abstraction. There are primitives defined (in predicate logic) for each level of abstraction; as the levels are traversed, we gradually ''flesh out'' the details of a plan for achieving the goal. ABSTRIPS relies heavily on the use of a theorem prover; hence, it is representative of much current AI computation. An example of its output is given in Figure 17.

```
Franz Lisp, Opus 38.92
-> [load abstrips.lsp]
t
-> criticality level: 4
skeleton plan : ((goal c))
criticality level: 3
skeleton plan : ((get-slippers d)
(give-slippers DOG ME) (goal c))
criticality level: 2
skeleton plan : ((gothrudoor c b DOG)
(gothrudoor b a DOG)
(gothrudoor a d DOG)
(get-slippers d)
(gothrudoor d a DOG)
(gothrudoor a b DOG)
(gothrudoor b c DOG)
(give-slippers DOG ME) (goal c))
criticality level: 1
skeleton plan : ((gothrudoor c b DOG)
(gothrudoor b a DOG)
(pushopen a d)
(gothrudoor a d DOG)
(get-slippers d)
(gothrudoor d a DOG)
(gothrudoor a b DOG)
(gothrudoor b c DOG)
(give-slippers DOG ME)
(goal c))
((gothrudoor c b DOG)
(gothrudoor b a DOG)
(pushopen a d)
(gothrudoor a d DOG)
(get-slippers d)
(gothrudoor d a DOG)
(gothrudoor a b DOG)
(gothrudoor b c DOG)
(give-slippers DOG ME) (goal c))
->
```

Figure 17: ABSTRIPS Output

The problem in our example was to have a dog fetch your slippers from another room. This problem takes about 15 minutes to plan on a VAX-11/750; the implementation makes heavy use of recursion and maintains several large lists.

The size of the Franz executable (from the UNIX `size` command) is 139,264(text) + 511,488(data). The data on memory usage was obtained by using the UNIX system's ability to create a *core dump* of a process's address space; since the text segment is read-only, only the data and stack segments are dumped. Sending the SIGQUIT signal to a process causes a core dump; this was done at the following points in the execution of

the ABSTRIPS planner.

1. When the LISP interpreter was started. This gives us a baseline value, with no program loaded and no code executed. The core dump occupied 528,384 bytes.
2. Immediately after ABSTRIPS was loaded. This tells us how much of the address space change is due to storage of the ABSTRIPS program. The core dump occupied 556,032 bytes; a bytewise comparison with the previous dump showed that 56,937 bytes had changed.
3. Immediately after ABSTRIPS execution is terminated. This tells us how much of the address space has changed during execution. The core dump occupied 613,376 bytes, and differed from the previous dump at 77,910 bytes. The difference between this dump and the first dump was a total of 123,942 bytes changed. No garbage collection was announced.

An important issue is the locality of reference; our measurement programs for the "copy-on-write" fork performance showed that we could write every page by writing one byte on each page. The byte comparison routine delivers addresses where it found differences between two files; the difference in bytes could then be measured by piping the output to "`wc -l`;" if we divide each address by the pagesize (512 on the VAX) and pass the results to "`uniq | wc -l`," we can find the number of pages that have changed: in this case 270 of the 1,198 (=613,376/512) pages changed, for a write fraction of 0.23.

4.2.5.2. GNU Emacs

GNU Emacs provides a facility to dump the currently executing image into an executable file. When this file is executed, the state of the Emacs interpreter is restored to the state it had when the `(dump-emacs)` was invoked. We took the following measurements on the VAX/11-750[12]. The size of the GNU Emacs editor we measured was 437,248(text) + 208,896(data), determined with `size`. We sought an example program

[12] The results for GNU Emacs were checked on the workstations, and they are consistent. For the "Towers of Hanoi" problem discussed below, the fraction of the data altered by the program was 0.30 on the 3B2 and 0.48 on the HP9000. Much of the difference is due to what features the runnable Emacs is pre-loaded with; the VAX executable has large amounts of pre-loaded information, which is read-only.

which had the sort of behavior (computation-oriented) that we desired. We based our desire for computationally-intensive examples on the observation that as heavy resource users, these programs would demonstrate the greatest effects from an optimization. GNU Emacs provides a library of LISP code; one routine provides a graphic solution of the classic "Towers of Hanoi" problem. We ran the GNU LISP interpreter on the following input:

```
(dump-emacs "pre-hanoi" "/usr/local/emacs" )
(hanoi 10)
(dump-emacs "post-hanoi" "/usr/local/emacs" )
```

(As might be expected, this requires patience at 9600 bits per second!) The interpreter emitted several messages to the effect that it was performing garbage collection.

At the completion of the computation, we performed a bytewise comparison on the two dump files:

```
$ ls -l post-hanoi pre-hanoi
-rwxr-xr-x  1 jms      phd         851968 Oct 27 08:25 post-hanoi
-rwxr-xr-x  1 jms      phd         737280 Oct 26 16:01 pre-hanoi
```

which showed that 183,312 bytes had changed, which for the computed data segment size of 414,720 (=851,968-437,248) is slightly less than thirty-five percent of the dump; that is, almost the same percentage we had observed with Franz Lisp and ABSTRIPS. Several times during the computation and in the `dump-emacs` function the garbage collector was run. Thus the amount which appears to have changed may include parts which did *not* change but were relocated and thus appear to have changed. It also compacted storage which appeared to be changed (since newly-allocated storage is considered changed from the previous non-allocated storage). The important point is that these changes would be seen by a page-management mechanism in either case.

4.2.6. Conclusions about copy-on-write

"Copy-on-write" paging strategies for address space inheritance have been shown to be effective in reducing the real time required to perform UNIX *fork()* operations. This qualitative assessment is based on the quantitative data we gathered and analyzed. For large processes, the time required is proportional to the fraction of write references, so

that a child process which updates half (0.5) of its address space will spend half the time doing copying that a child process which updates all (1.0) of its address space will. For a pair of interpreters with large address spaces, we showed that the portion of the address space changed from process startup until process termination was small, typically less than 0.5. These measurements concur with those of Zayas [Zayas1987a], who measured program behavior in an Accent environment, and confirm the desirable properties observed of a similar scheme for fast state transfers to remote systems in the V [Theimer1985a] system[13].

Thus, if these interpreters or programs which behave similarly were to *fork()* child processes which executed tasks similar to those described, a reduction of 50 percent or more of the system time devoted to copying data might be achieved. This confirms that the scheme for remote *fork()* using "lazy" copying has considerable merit.

This reduction in copying also reduces the amount of swap space required, reduces the amount of time spent swapping, increases the number of processes which can be run without paging, and decreases the *cost* of context switches (where the cost of paging out the written pages and the paging in of pages which are only read and have not been modified is included). Thus the advantages of the text table are extended to unmodified pages (or viewed another way UNIX gains via "copy-on-write" the ability to *eliminate* the text table and improved *fork()* performance). With respect to these page management strategies, note that TENEX [Bobrow1972a] had these advantages ten years earlier and needed neither a distinguished text table nor the confusion of two varieties of *fork()*.

The cost figures we present should be representative of a shared memory configuration of equivalent processor technology. The fact that we have provided a methodology for gathering such measurements ensures that the techniques are portable, even if the measurements themselves are not.

[13] See the description in Smith's survey paper [Smith1988b].

4.3. Remote *fork()*

There is more overhead associated with the distributed case, due to the increased costs of copying. We describe a method of implementing a distributed *fork()* operation in Smith and Ioannidis [Smith1989a]. A process successfully executing a *fork()* operation generates two copies of its address space; these are often distinguished as *parent* and *child* by the return value of the *fork()* call. If the *child* process continues its execution with the containing address space located on a processor different from the *parent* process, we have achieved a ''remote fork.''

By distinguishing between the state saving activity and the state transfer activity, we were able to measure and refine the performance of each activity independently. Once the design and initial implementation were complete, we analyzed the performance, and reimplemented pieces of the system (on several different machine architectures) to improve the response time. This improvement was dramatic: from about 7 seconds of real time on the HP9000 and the 3B2, to less than 1 second on the HPs and Suns using NFS. The major savings came from reducing the execution time devoted to copying state information from point to point.

4.3.1. Further process migration ideas

As the major cost of process migration is copying [Zayas1987a], attempts have been made to reduce this overhead. More sophisticated migration schemes, using ''on-demand'' state management techniques have been constructed [Theimer1985a]. Most programs exhibit *locality of reference;* in particular symbolic computations which use large amounts of system resources [Smith1988a]. These computations are representative of those that present the greatest opportunity [Leland1986a] for applying process migration to load-balancing across multiple processors. In early computer systems, the notion of a relocatable (position-independent) module of executable code improved the degree of multiprocessing. Relocatable code allowed multiprocessing systems to exploit available memory more effectively. This exploitation increased the degree of multiprogramming, achieving an increase in throughput. Likewise, *relocatable processes* give operating systems the capability to exploit the availability of multiple processors. This exploitation can result in improved throughput, improved response time, or both. What makes a process *relocatable* is a description of its state which can be used to continue the

computation elsewhere. This description is often achieved via a copy, thus forcing the expensive copying operations. Measured in execution time, the cost of copying can be reduced by faster networks or data transfer software. Our implementation of *rfork()* shows that this approach can be effective. However, for any data transfer scheme there is a limit on its performance. This limit is imposed by the combination of the transfer hardware and the software used to access it. Also, although the mapping between data volume and execution time may increase in steps rather than smoothly, *more data implies more time*. Thus, for a limited data transfer rate, we should seek methods of reducing data copying.

Several ideas suggest themselves:

1. *Be lazy*. This describes the approaches of Zayas and Theimer; they took advantage of locality to reduce copying or its effect on program execution speeds. ``Demand-paging'' has illustrated the effectiveness of lazy copying in computer systems.
2. *Encode Symbolically*. The state can be encoded symbolically. Symbolic encoding can be achieved by use of interpreters or compiled interpretative languages. When the execution of a process is halted, the program and the state of the interpreter are re-represented in a symbolic form. This symbolic form is then passed to another interpreter, which can restart the program from the ``symbolic checkpoint.'' This representation may also be more compact than the running program. Note that such machine- independent representations offer the only hope for true ``heterogeneous'' process migration.
3. *Compress*. Another possibility for re-representation is *compression*, where the encoded state is produced by applying a data compression algorithm [Lelewer1987a] to the saved state. When the process is restarted, the state is uncompressed and used to recreate the running process.

Schemes 2 and 3 pay a computational cost in encoding and decoding. Thus, there is a performance tradeoff between the cost of copying and the cost of encoding. For straight-forward compression techniques, the cost of encoding can be calculated using the size of the input state. For ``symbolic checkpoints,'' cost estimates are less predicatable from the size of the process, but can easily be made using the symbol table which is necessary for the encoding to take place. Copying costs can be estimated using the techniques we have described in this thesis, so that decisions can be made by a dynamic migration

manager. These decisions, about whether to migrate or not, can help in load-balancing.

4.4. Disk Response Time

Referenced pages will not always be available in memory, thus disk access (or network access, to be discussed later) may be needed for a reference to be satisfied. One difficulty with simulation, or with "toy" implementations is that the quantitative data necessary for accurate response time evaluation are not used in the simulation. To make our results more accessible to practitioners, we have used data on disk response times [Johnson1987a] which was gathered on a running UNIX system, operating the Digital Equipment Corporation™ (DEC) RA81 drives whose characteristics are summarized in the table,

DEC RA81 Disk Drive Characteristics	
Cylinders	2516
Transfer Rate	2.2 megabytes/second
Rotational Speed	3600 rpm
Average Rotational Latency	8.33 milliseconds
Head Switch Latency	6 milliseconds
Average Seek	28 milliseconds
One Cylinder Seek	7 milliseconds
Maximum Seek	50 milliseconds

attached to a DEC UDA50 controller. Under the load conditions described by Johnson, Smith, and Wilson as "normal," the average number of ticks per response was 2.1, thus the average response time using a tick of 1/60 second was $ 2.1 cdot (1/60) $, or 35 milliseconds. From their data analysis, it can be seen that this number is constant across drives, interfaces, transfer sizes, and transfer start time. It varies between reads and writes, with writes taking longer, and it varies with the location on the drive. Writes take longer because they are bunched together at times when UNIX flushes its buffer cache,

thereby lengthening the time spent in the drive's request queue. The response time varies with the location on the drive due to hot-spots found at the *i-lists* of file systems found on the drive. Both of these variations are due to the nature of the UNIX file system, and thus are not significant for our discussion.

4.5. Network Response Time

To gather information on the response time for network page requests, the program (''netrand.c'') was written. Since it is short, a source listing is provided as Appendix II. The experiment was to operate the program on two files, one on local disk, and the other on an NFS-mounted file system. The system employed for testing was a Hewlett-Packard HP9000/350 with 8 megabytes of main memory and an HP7945 70 megabyte hard disk, running HP-UX™ 6.0.

The UNIX buffer cache mechanism was frustrated by copying a large (2 megabyte) file previous to running the tests. Three executions of the test were run on each file; for the purposes of benchmarking we ran the program with an argument of 400 blocks. Since the randomization frustrates the buffer cache mechanism to some degree, the buffer cache has a significant effect by the third execution. The results of running the program on the local file were 10.68, 10.24, and 9.74 seconds of elapsed time for the three runs. For the remote file (the server is a machine of the same type, accessed over a lightly-loaded 10Mbit Ethernet) the results were 15.42, 15.02, and 13.32 seconds of elapsed time for the three runs. The improvement in performance seen as the runs progressed is due to the success of the UNIX buffer cache in retaining recently-read blocks. The network access seems to indicate a penalty of about a factor of 1.5 for network access of pages, which is encouraging. The time per page (using the worst case, where the cache is flushed) gives us a per-page time of $\frac{15.42}{400}$ seconds, or about 39 milliseconds per page. This works out to about 26 pages per second for 4K pages. This cost compares unfavorably with the cost of a local page copy, which is about 1 millisecond on this machine.

4.6. Sibling Elimination

A remaining source of overhead is the cost of eliminating unwanted computations; we have speculated, selected, and now we must eliminate. What will this cost us in time? A program, *do_elim.c*, is presented in Appendix XII. It was constructed to address the various factors which might influence the time involved in eliminating processes. While the details of the construction are evident in the program, some discussion of the goals and philosophy is in order. The goal was to give us some idea of the cost in execution time attributable to sibling elimination which would not be paid in the case where the fastest alternative was selected "at random." This helps us to estimate $\tau(overhead)$.

The philosophy was to use a highly-tuned existing operating system, in this case Hewlett-Packard's HP-UX 6.0 implementation of UNIX System V. The idea behind that is that we can experiment with various tunable parameters of interest, while taking account of details which a simulation might otherwise ignore. In addition, the numbers should be close, certainly less than an order of magnitude away, from values gathered in an implementation. We gathered data for **real, user,** and **system** times, obtained from the UNIX *times()* system call. The UNIX timing facility is not particularly accurate, but we applied techniques which should, in the average case, remove much of the error content from the data.

The basic design of the experiment was to create some number of processes, have them each do something after spawning, and then eliminate them. Only the elimination portion of the experiment is bracketed with timing requests. Elimination is done by means of the UNIX *kill()* system call, which sends a small (less than a byte) amount of data from a *sender* to a *receiver* in the form of a *signal*. The *signal* causes the receiver to execute some signal-handling action, which can include terminating, ignoring the signal, or handling the signal in some specialized fashion. The processes in our experiment were set up in a fashion that ensured process termination upon receipt of a signal. Bach [Bach1986a] provides discussion of these mechanisms in such detail that further detail here is unnecessary.

The factors we chose to examine were:

Groups: UNIX provides a facility to send a signal to (1) a single process, or (2) a group of processes, selectable by virtue of either the owner's *user id* or the *process group*. Use of the process group signaling facility allowed us to emulate the

effects of a multicast message-sending facility, so that we could evaluate the effects of point-to-point elimination versus multicast.

Files: Among the items of system-maintained state associated with a UNIX process are a number of open files and their associated data structures. Upon exiting, these files must be closed, which involves deallocating them, flushing some buffers, etc. Thus, varying the open files should vary the system state maintained by a process which is to be deleted.

Size: As we saw the effect of virtual memory copying on response time earlier in the thesis, it may have some impact on process deletion, in particular the deallocation of allocated pages and page descriptors. Varying the size gives the best estimate of the impact on response time, as we saw earlier.

Work: The spawned subprocesses either loop *sleep()*ing, or they alternate between sleeping and iterating in an empty *for()* loop. The latter case tries to estimate the effect of the UNIX process scheduling policy on processes; in particular, if the wait for a working process would be longer than the wait for a sleeping process.

Asynch: After signaling the spawned subprocesses with *kill()*, we can either wait for them to complete inside the timing block, or we can exit the timing block, to emulate a situation where we don't wait for the processes to die, but just signal that they should die, and then continue. The waiting case is synchronous, and the other is *asynchronous*. With a large degree of parallelism present, asynchronous elimination seemed a better policy, as the hardware parallelism could be employed to speed up the elimination.

Dirty: As with the size of allocated memory, we earlier discovered that the write fraction had a determining effect on response time when coupled with memory

Procs: This varies the number of spawned sub-processes. The idea is to see how the increase in sibling elimination costs varies with both the policy employed for elimination, and the number of processes the policy is applied to.

The measurement program, *do_elim.c*, was run using the shell script in Appendix XII. The numbers were extracted, and entered into an S [Becker1984a] dataset for analysis. The statistics system was used as a tool for data analysis, in the sense that various parameters were related to the **real** and **sys** times they exhibited, to determine the importance

of their effect on the response time. The next two sections present the data in graphical form, with a short discussion and explanation following each figure. The first of these sections presents the **real** time measured, and the second section shows the **sys** time measured. The point of **real** time is that it's the best measure of response time; **sys** time tells us how much effort the system is exerting on our behalf. In response to well-known problems with the granularity of the UNIX clock scheme, we ran the *creation- timed deletion- cleanup* loop 100 times (by setting REP_COUNT to 100 in *do_elim.c*) for each variety of *do_elim* invocation. This has two effects. First, it gives us enough magnitude in the data so that comparisons can be made between the different test inputs. Second, the timing errors in such a loop can be lesser or greater than the correct time. With repetitions, the idea is that the errors will in effect *cancel*, so that the repetitions will in the average case purify the data, much like oversampling techniques in audio. Of course, in the worst case, correlated or systematic errors, there is the possibility that the results will be all noise. This possibility is refuted later.

4.6.1. Real Time

To measure the effects of the different variables on the execution time (**real** time, in UNIX jargon) we plotted the real time values as a function of the variable values. The most useful technique we have found for representing this data is the *boxplot;* it provides much more information than an **X-Y** plot for this type of data. For a given **x** value, the box defines the middle 50 percent of the data, the horizontal line inside the box is the median, and the bar at the end of the dashed line marks the nearest value not beyond some standard range (in this case, 1.5·(*inter-quartile range*)) from the quartiles. Points outside these ranges (``outliers``) are shown individually. Details of *boxplot* presentation can be found in Chambers, *et al.* [Chambers1983a]

The first plot, figure 18, indicates that the use of the process group mechanism reduces the execution time devoted to sibling elimination. While the median improvement is not great, the larger top half of the left hand box for not killing by groups indicates that for the slowest half of the times, group signaling had a more significant effect.

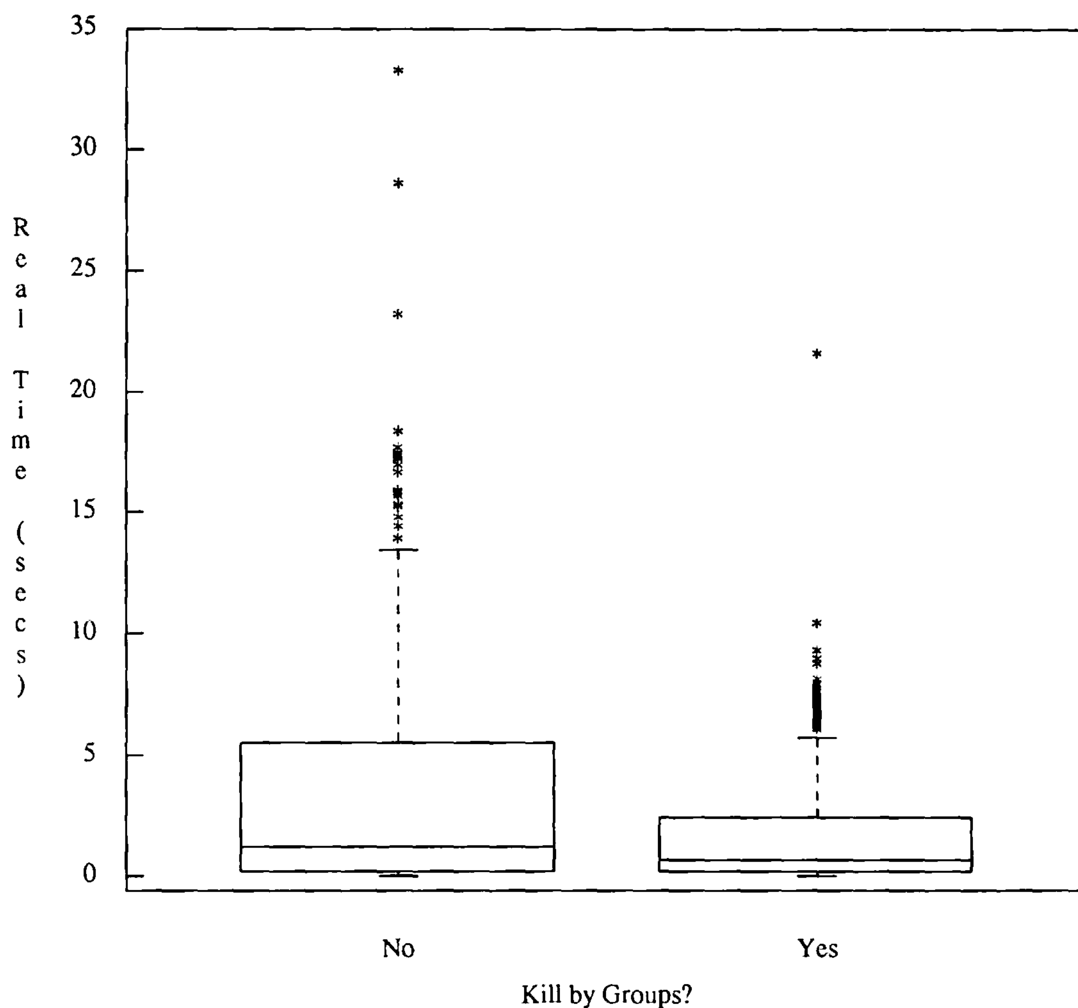


Figure 18: Effect of signaling process groups on real time
(For 100 repetitions)

This is an intuitive result, since broadcast should be significantly cheaper in wall clock time than a serial sending of the "messages."

Figure 19 shows the effect on real time of the number of open files: it appears that the amount of system state which must be changed when these are closed on process termination is not significant in its effect on sibling elimination times.

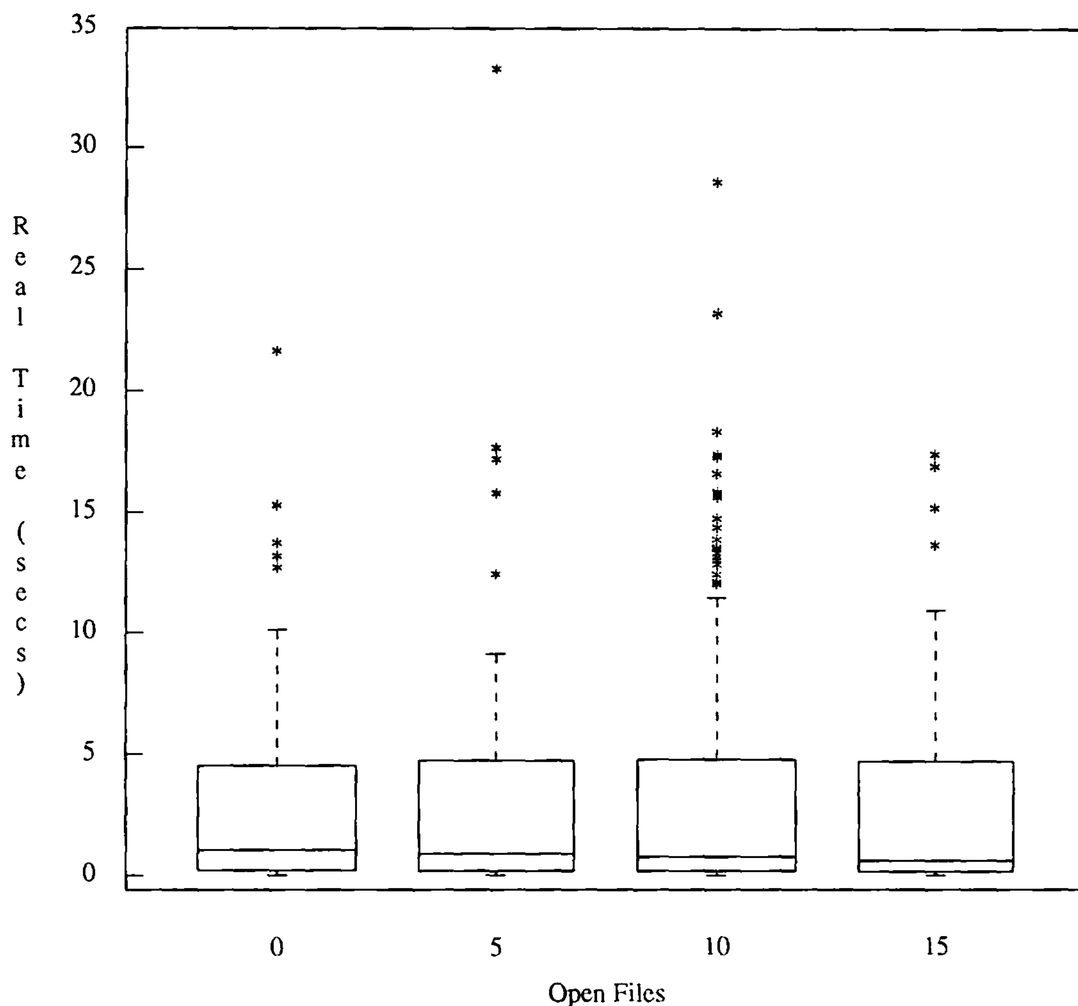


Figure 19: Effect of number of open files on real time
(For 100 repetitions)

Figure 20 shows the effect of performing the elimination synchronously (Asynch=0) versus asynchronously (Asynch=1). There appears to be no difference in the real time required. This is surprising, as intuitively, not waiting for something to happen should be faster than waiting for something to happen. There are two possibilities: (1) the effect is too insignificant to be discerned against the measurement error (a real possibility, given the granularity of the UNIX clock facility) or (2) the experimental apparatus is flawed. We address this counterintuitive result later in the thesis.

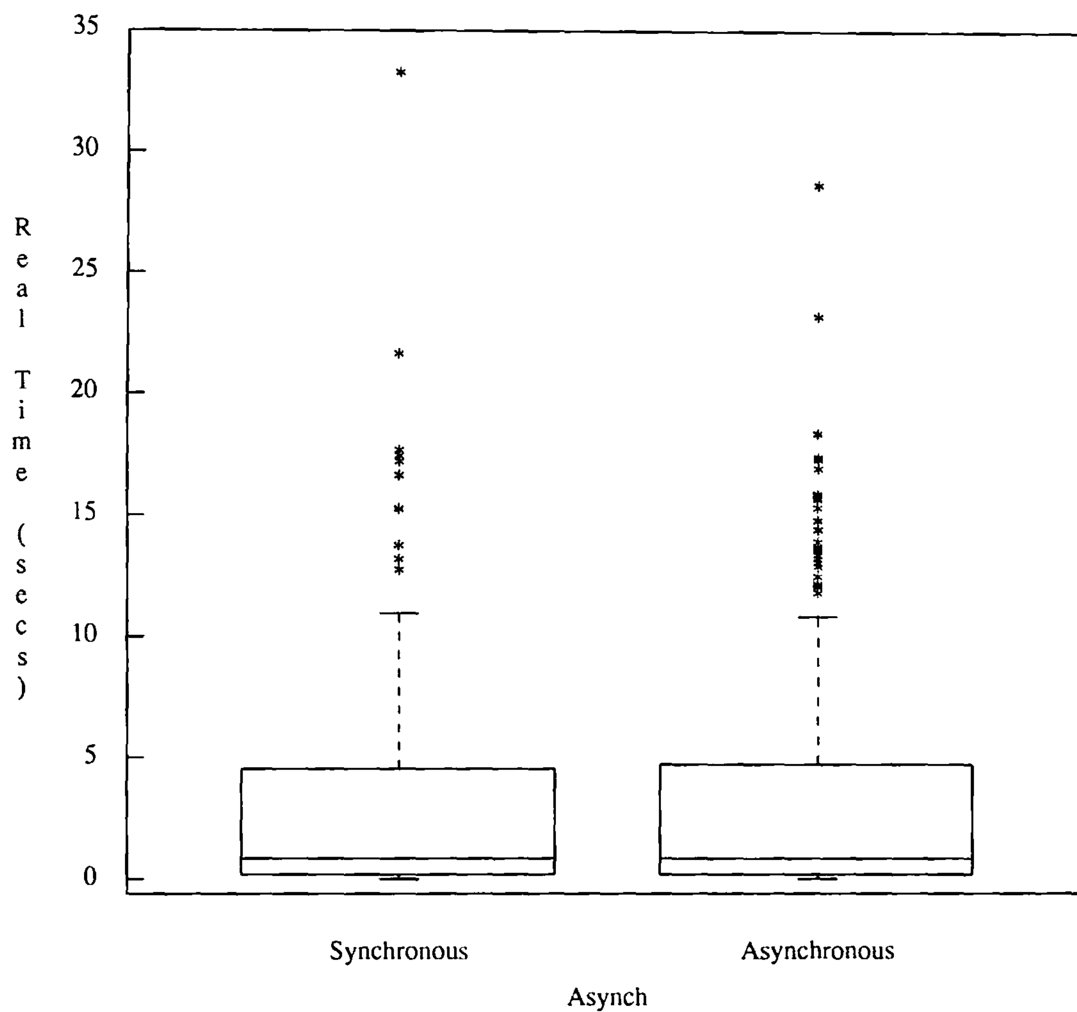


Figure 20: Effect of Asynchronous elimination on real time
(For 100 repetitions)

Figure 21 indicates that the effect of dirtying pages in the child ("dirtying" is done by the child, as it would be in truly concurrent execution) is negligible when measured in real time.

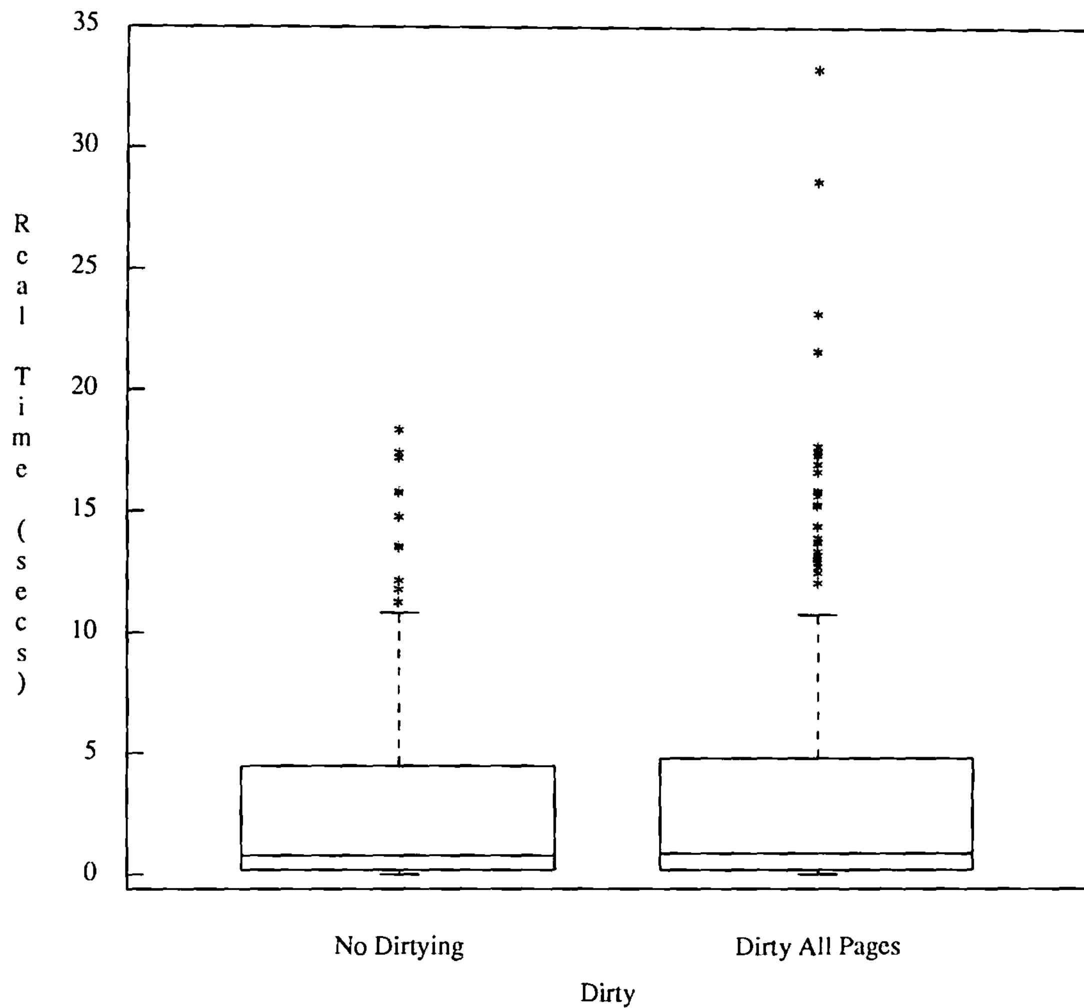


Figure 21: Effect of Dirtying child pages on real time
(For 100 repetitions)

Figure 22 indicates a slight effect on real time caused by the process behavior of the spawned children. The idea here was to see if the signals would take longer to propagate if the multiprocessing involved processing and not merely waiting for receipt of a signal. As one might imagine, it takes longer to eliminate busy children than it does sleeping children.

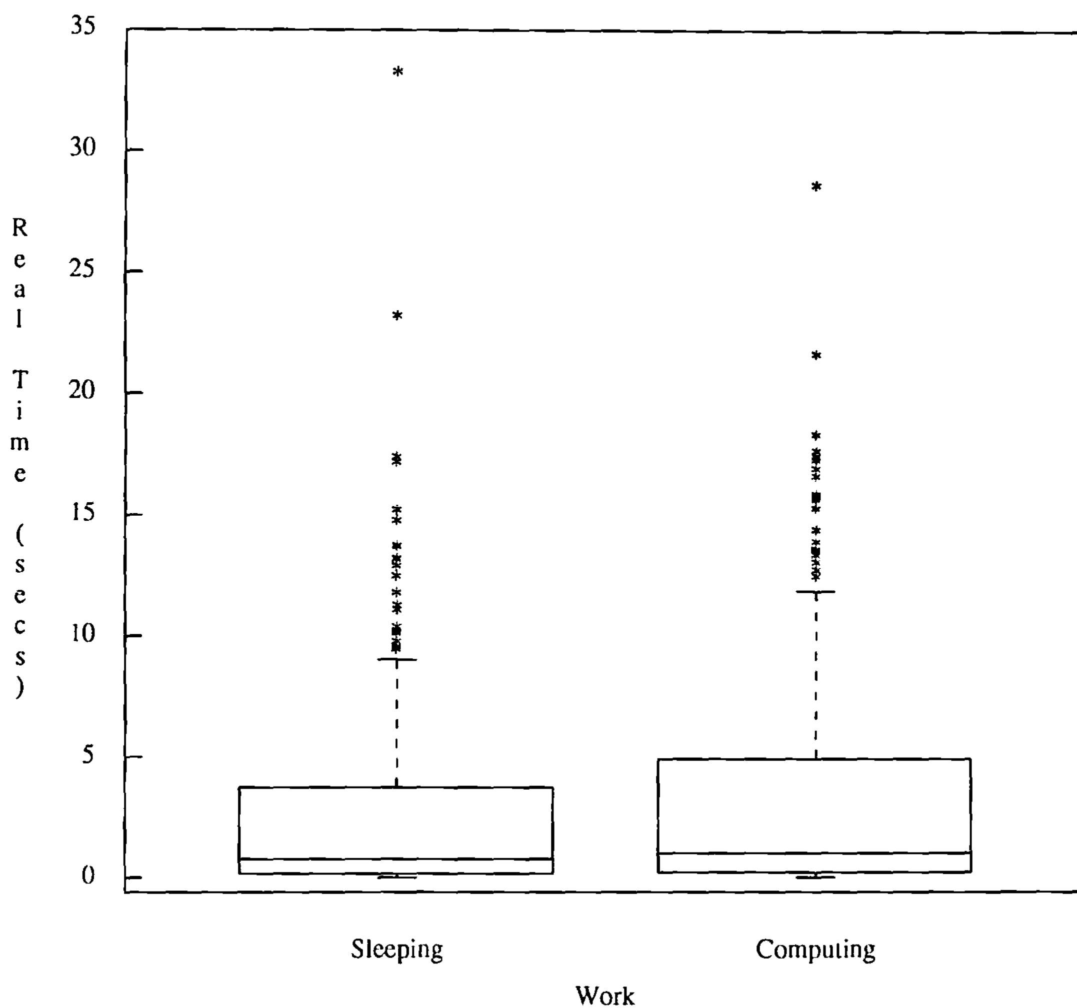


Figure 22: Effect of child work/sleep activity on real time
(For 100 repetitions)

Figure 23 indicates almost no effect contributed by the process size. There is a slight difference, most evident when the least (size=0) is compared with the greatest (size=100,000) where we see a slight increase in the max, median, and top 50 percent. The increase is so slight that it would be difficult to attribute a significant time cost to process size in sibling elimination.

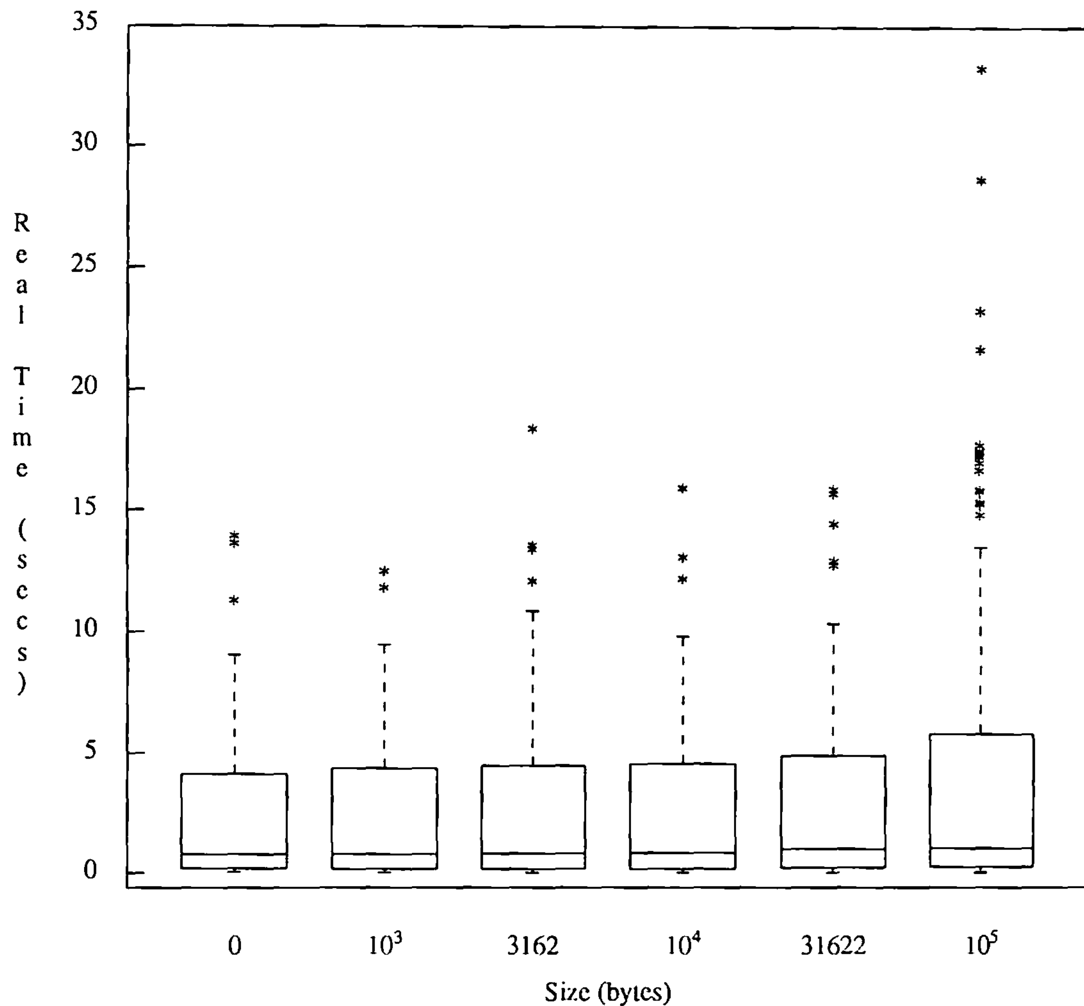


Figure 23: Effect of child process size on real time
(For 100 repetitions)

Figure 24 shows the effect the number of child processes (which are spawned and then eliminated) has on the measured real time. This plot indicates that the number of child processes has a significant effect on the required real time for sibling elimination, and that the cost increases in proportion to the number of children. The sample values for Procs formed an exponentially increasing sequence, giving rise to the shape of the curve drawn through the medians or the curve through the top "whiskers." These drawn curves differ slightly in their shape; 8 child processes is a jump in the value of the top whisker, where the curve through the medians is smoother. This discontinuity was not consistently reproducible. For example, real time and system time should be well-

correlated on these measurements, and yet a comparison of the graphs does not show the slight discontinuity in the curve connecting the top whiskers. Therefore it is probably measurement error; see the section below on errors in the measurements for a detailed discussion of the sources of such errors.

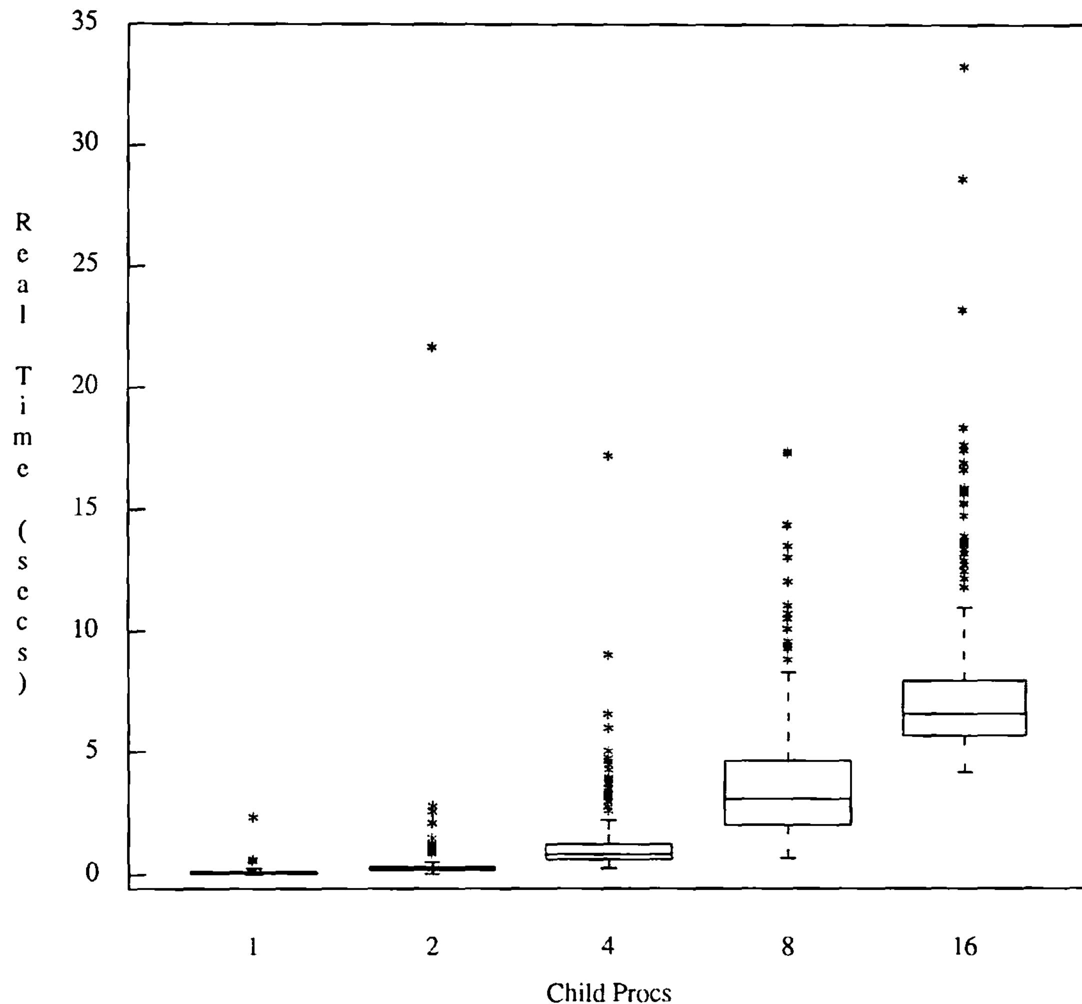


Figure 24: Effect of number of child processes on real time

(For 100 repetitions)

Thus, the data we have gathered indicate that there are two significant variables in the real time devoted to sibling elimination. These are (1) the use of process groups in communicating message information through signals (roughly equivalent to a multicast), and (2) the number of processes which are spawned and eliminated. Surprisingly, there was little effect seen from the use of asynchronous messaging. To refine our estimates of

measurement errors, we decided that plots of the system time required for each of the variables we analyzed the real time for were necessary. These are presented in the next section.

4.6.2. System Time

As in the last section, the time required for a multicast sibling elimination is reduced significantly when compared to serial message-sending. In light of the intuition and previous measurement results, this helps to confirm that the results are significant (and not noise) *if* the experimental apparatus is correct.

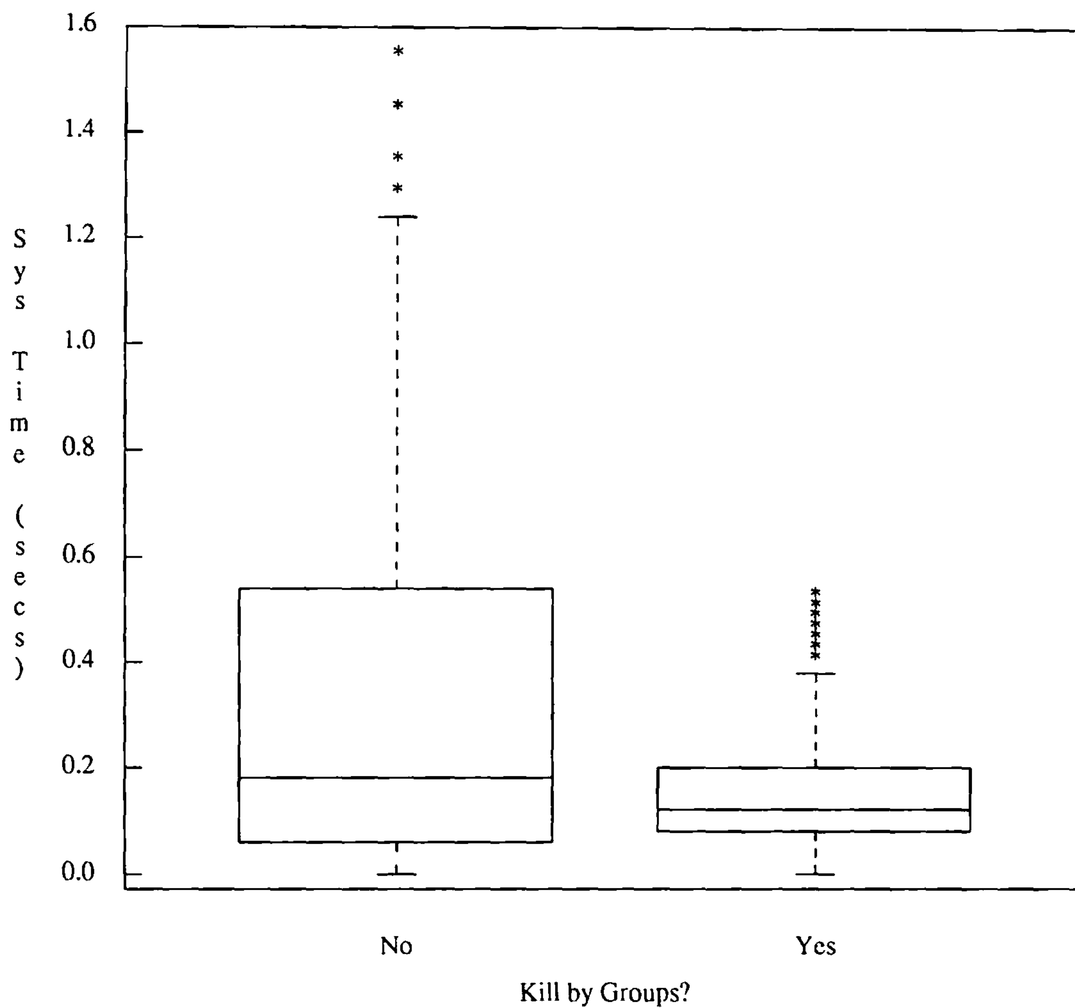


Figure 25: Effect of signaling process groups on sys time
(For 100 repetitions)

As before, the number of open files was examined as a contributor to the measured time values. There does not seem to be any contribution to the system time, as is evidenced by the plotted data of figure 26.

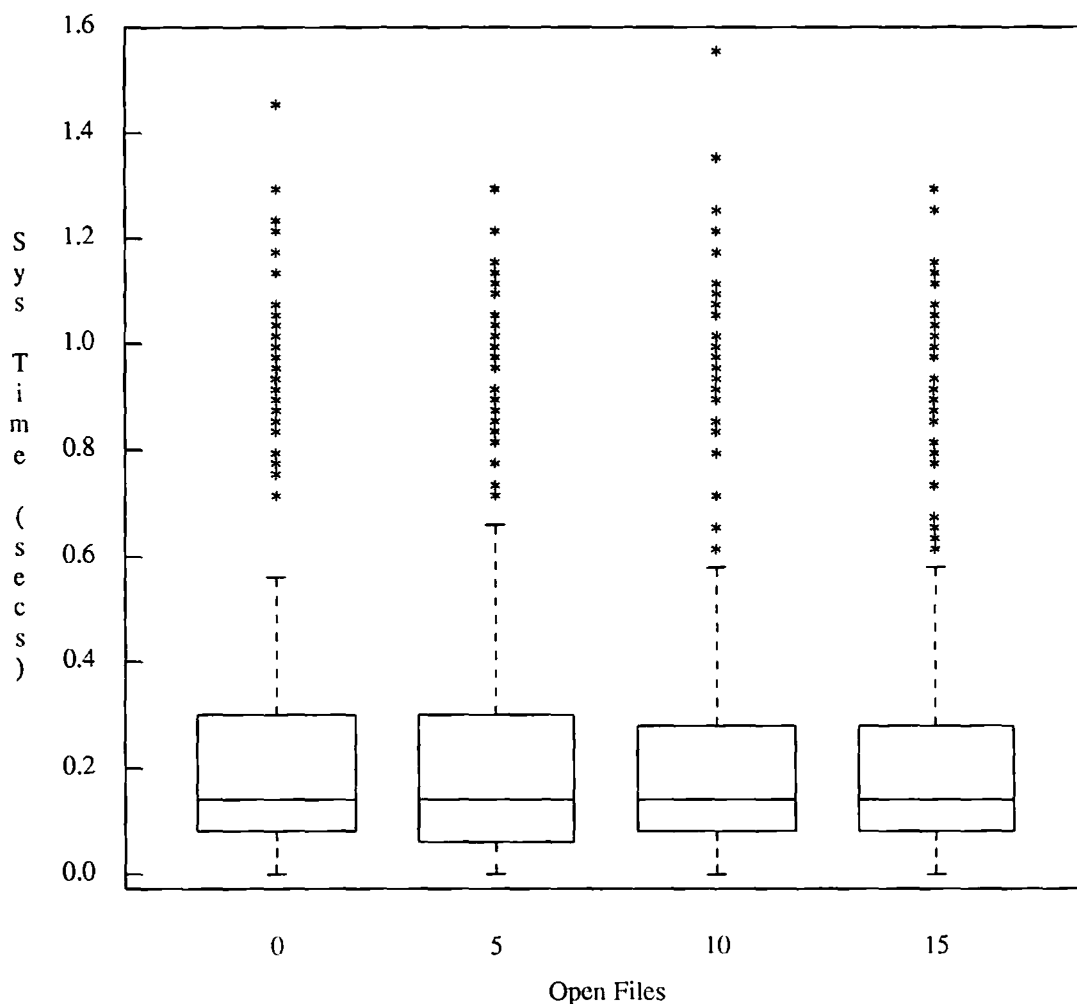


Figure 26: Effect of number of open files on system time
(For 100 repetitions)

Figure 27 shows the effect on system time of making the elimination process asynchronous. As in the analysis for real time, the results are counterintuitive; that is, there seems to be no difference. This seems to make more sense in the system time setting, as the system must do approximately the same work in each case; the exception might be in table searches which could be done on the basis of process group membership, rather than multiple searches for single process identifiers; however, the effect of this would probably be lost in the noise. Likewise, there is an increased number of system calls; their contribution appears to be minimal according to the plot.

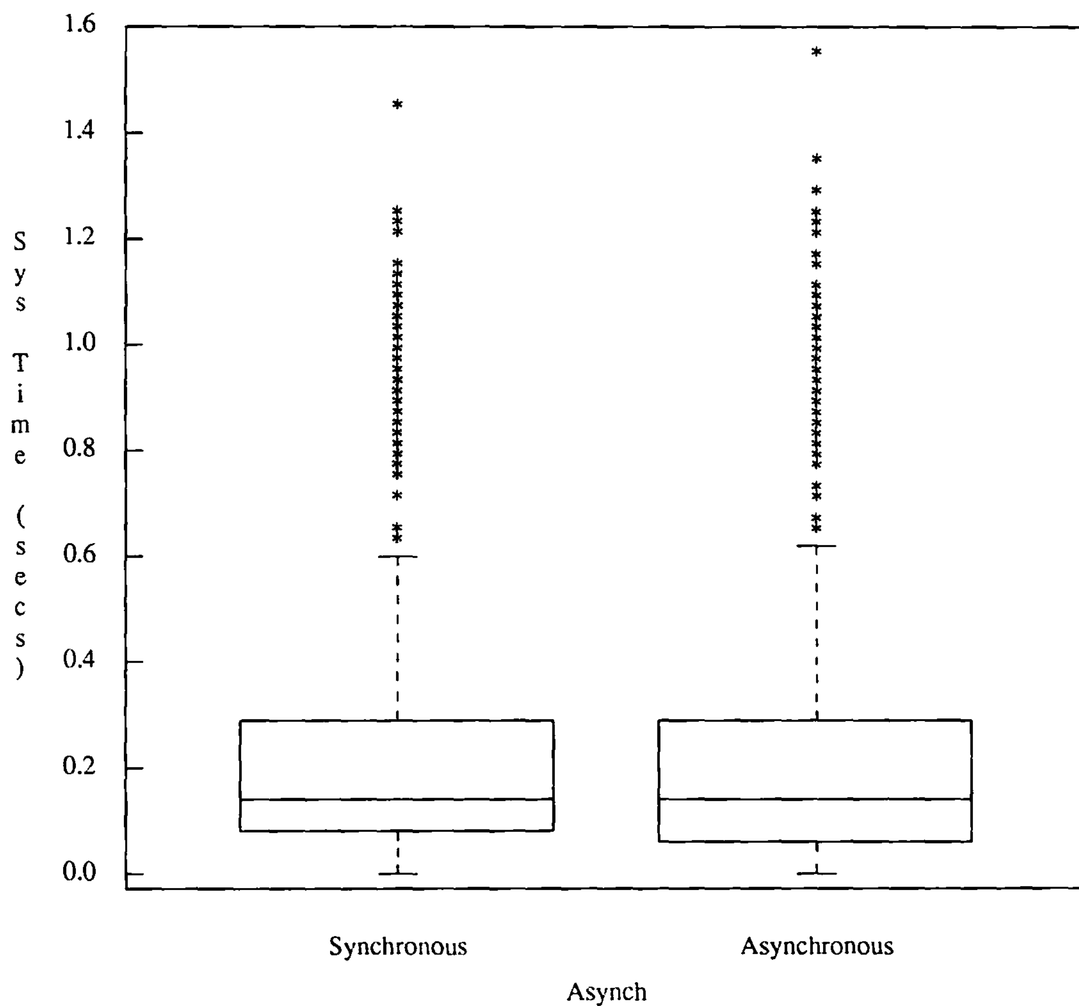


Figure 27: Effect of Asynchronous elimination on system time
(For 100 repetitions)

Figure 28 shows the effect of the child process's page updating behavior on the system time; as before, there is no observable difference.

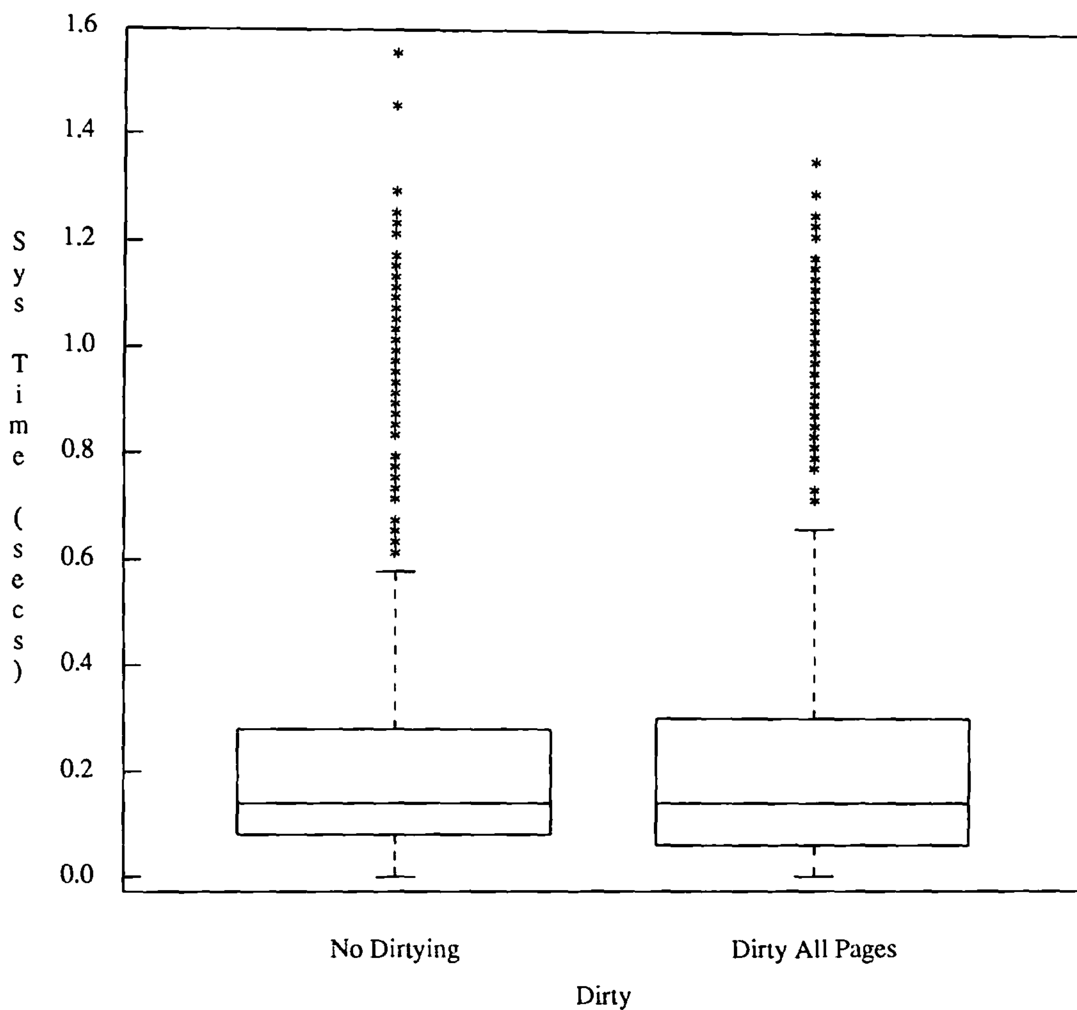


Figure 28: Effect of Dirtying child pages on sys time
(For 100 repetitions)

Figure 29 shows the effect of the child process's computational behavior on the system time; there is no observable difference.

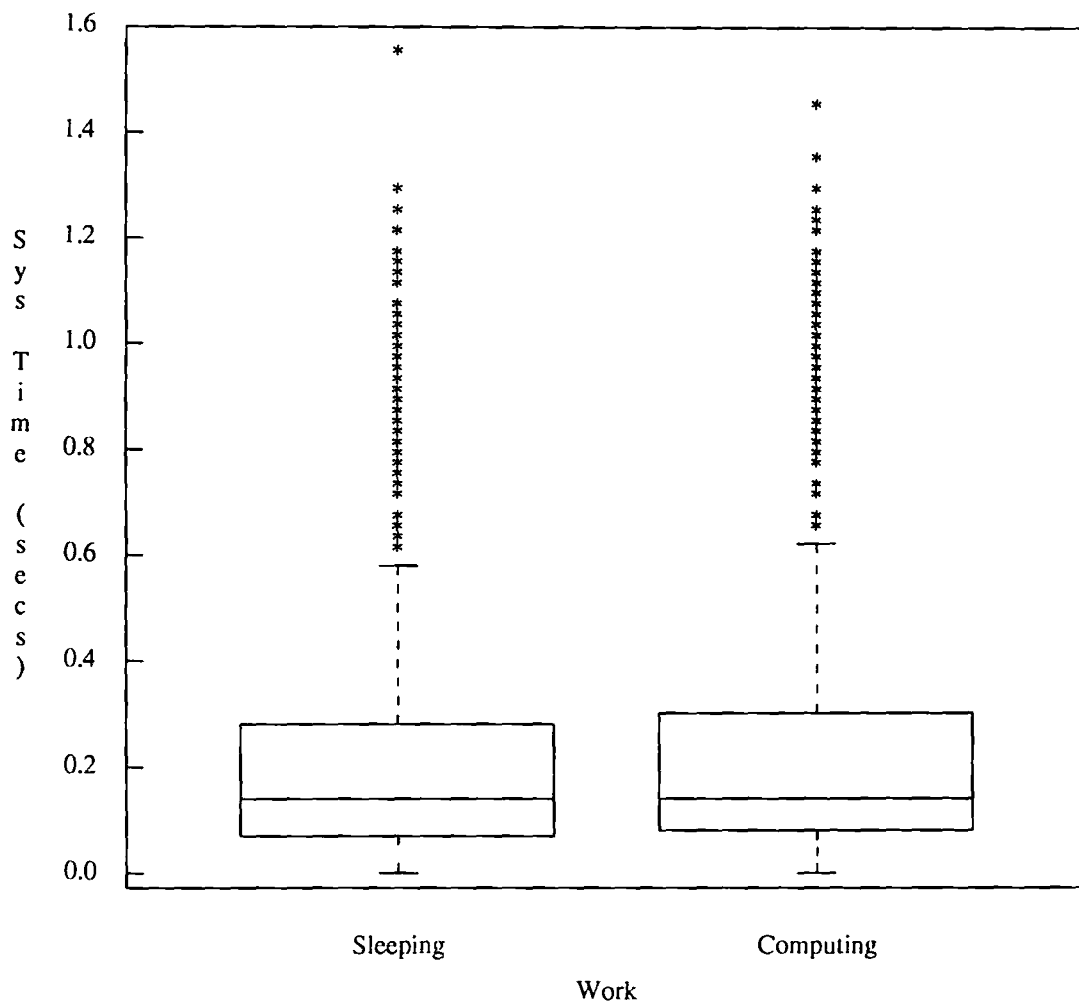


Figure 29: Effect of child work/sleep activity on system time
(For 100 repetitions)

Figure 30 shows the effect of the child process's allocated memory size on the system time; there is no observable difference, not even the slight effect we observed previously for real time. This may be due to the accounting scheme used for faulting pages; if the page fault time is charged to some kernel "process" the extra costs associated with page management (including deletion of dirty pages) may not be charged to the proper entity.

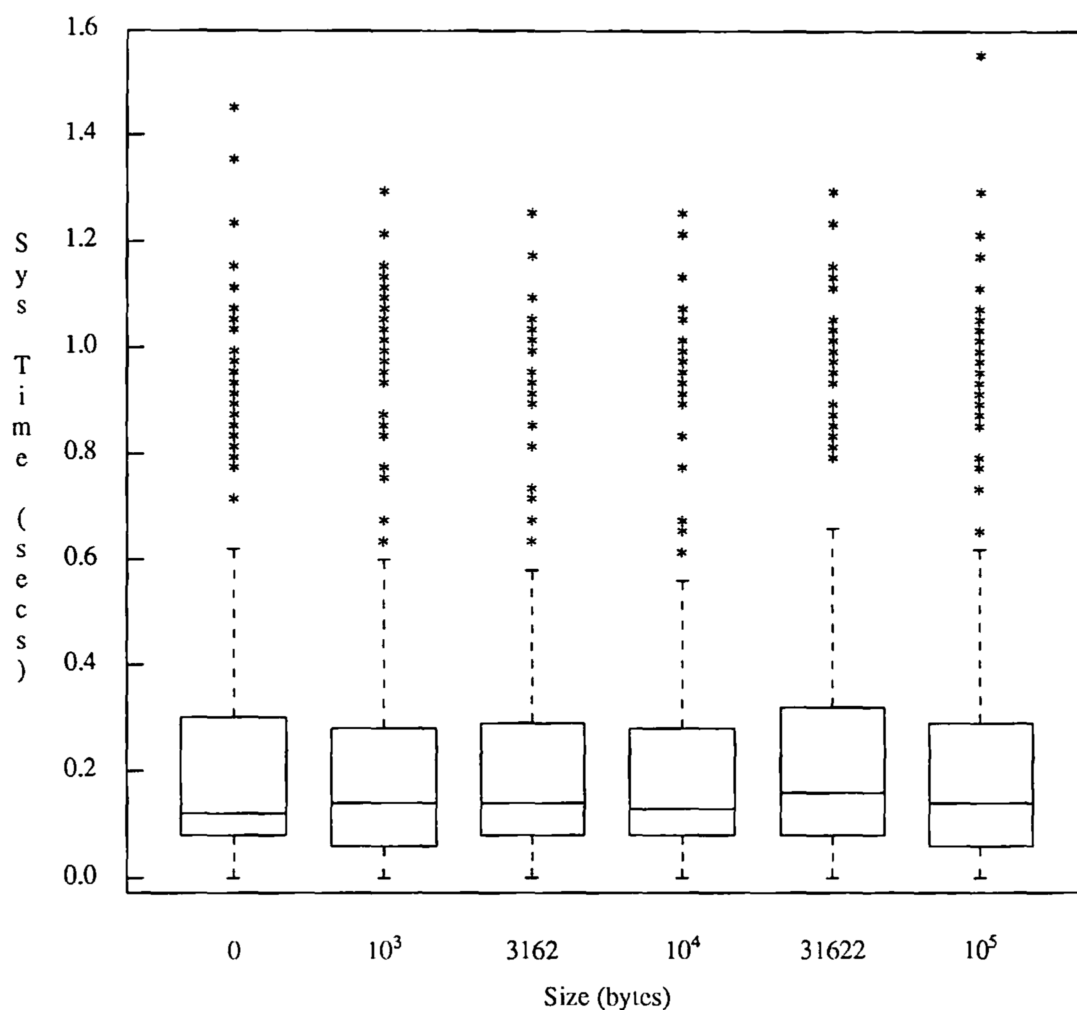


Figure 30: Effect of child process size on system time
(For 100 repetitions)

Figure 31 illustrates the effect the number of processes has on the system time. The increase is once again proportional to the number of processes; this increases our confidence in the behavior of the observed real time. It's interesting to observe that there is also a correlation between the number of processes and the dispersion of the system time values measured: much more so than our observations for real time. We have no explanation for this other than clock inaccuracy and the contribution of other variables. For example, the contribution of data from both values of a bimodally distributed variable such as Groups may cause an apparent dispersion. Also, since the system time values are always smaller than real time values (on a uniprocessor) the effect of clock granularity on

measurement accuracy will be more pronounced. Clock granularity as a source of error in measurement is discussed below.

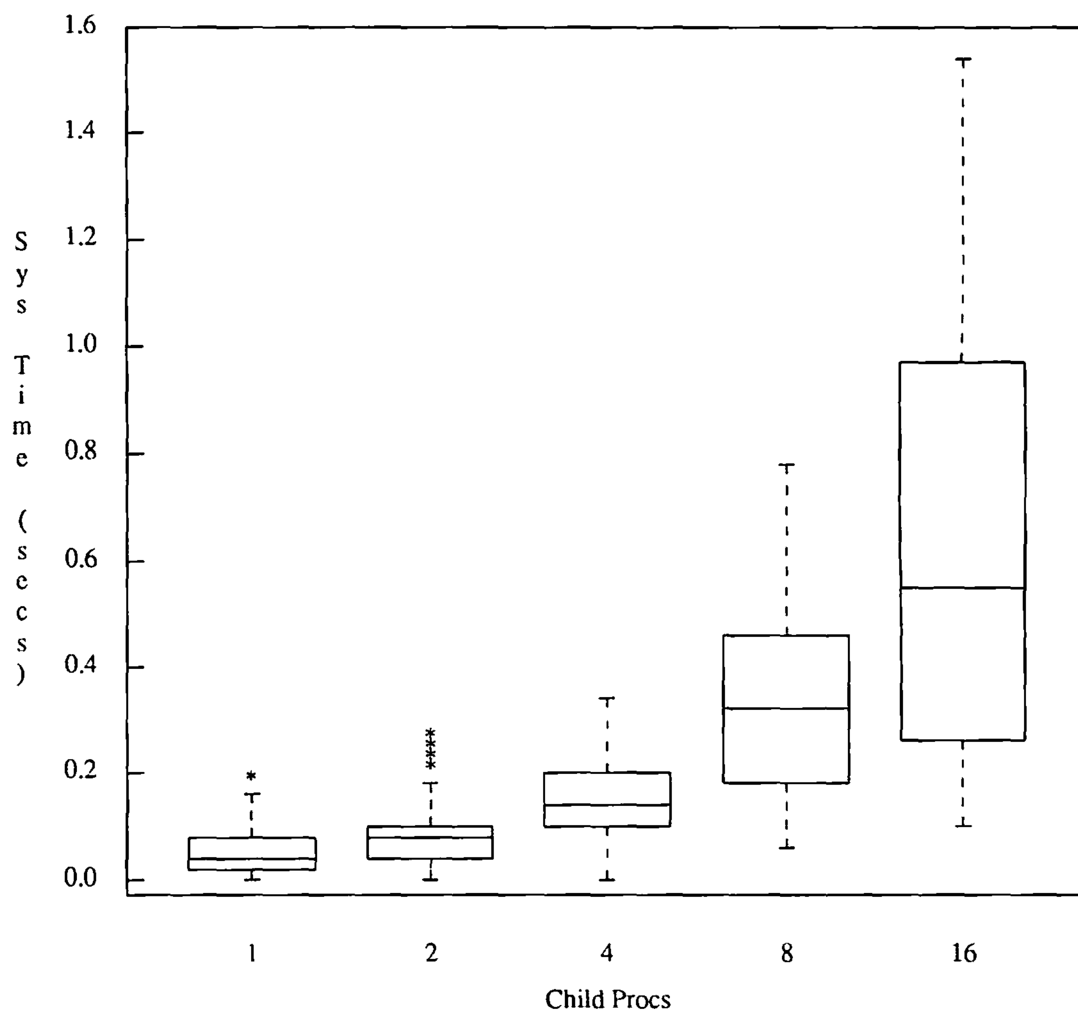


Figure 31: Effect of number of child processes on system time
(For 100 repetitions)

Thus, our observations of the system time correlate well with our observations of the real time. What this does is to confirm that where the data shows an influence, there exists an influence. The non-intuitive values for asynchronous elimination drove us to further examination of the data and the apparatus.

Since there was a clear correlation between the number of processes spawned and eliminated and the time required for both timing measurements, it seemed worthwhile to examine the most expensive case separately, to remove extraneous influences form the

data. In the next section, we examine the influences of the variables for the special case of 16 spawned and eliminated child processes.

4.6.3. Real Time, 16 Procs only

As in the less restricted set of measurements, the use of process groups seems to provide a significant improvement in real time performance. Figure 32 illustrates the magnitude of this improvement.

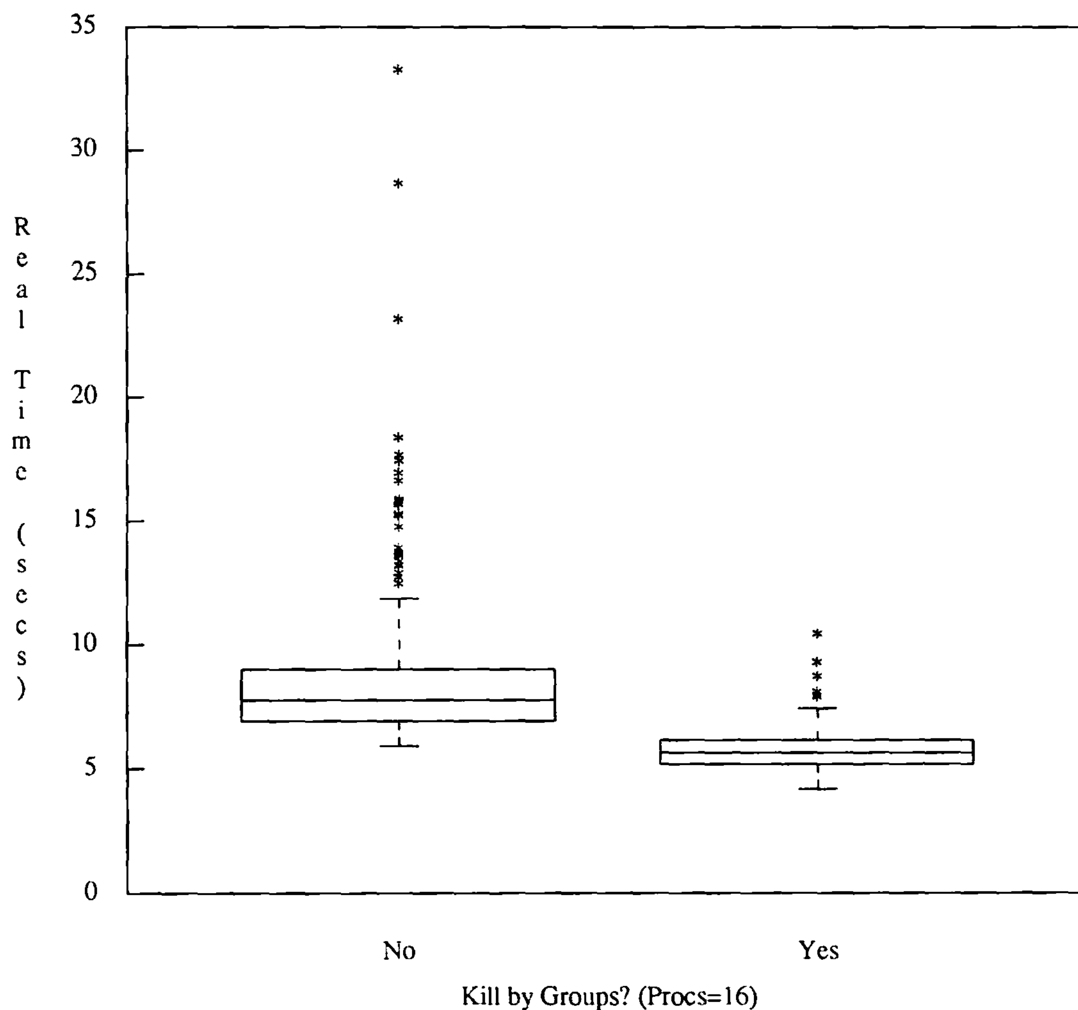


Figure 32: Effect of signaling process groups on real time, 16 procs (For 100 repetitions)

Figure 33 indicates a small contribution from the number of open files; this makes sense, as for the larger number of processes involved, the system has significantly more state to

manage. However, the contribution still appears slight: it is slight enough to be accounted for by errors, as discussed below as "Sources of Errors in Measurements."

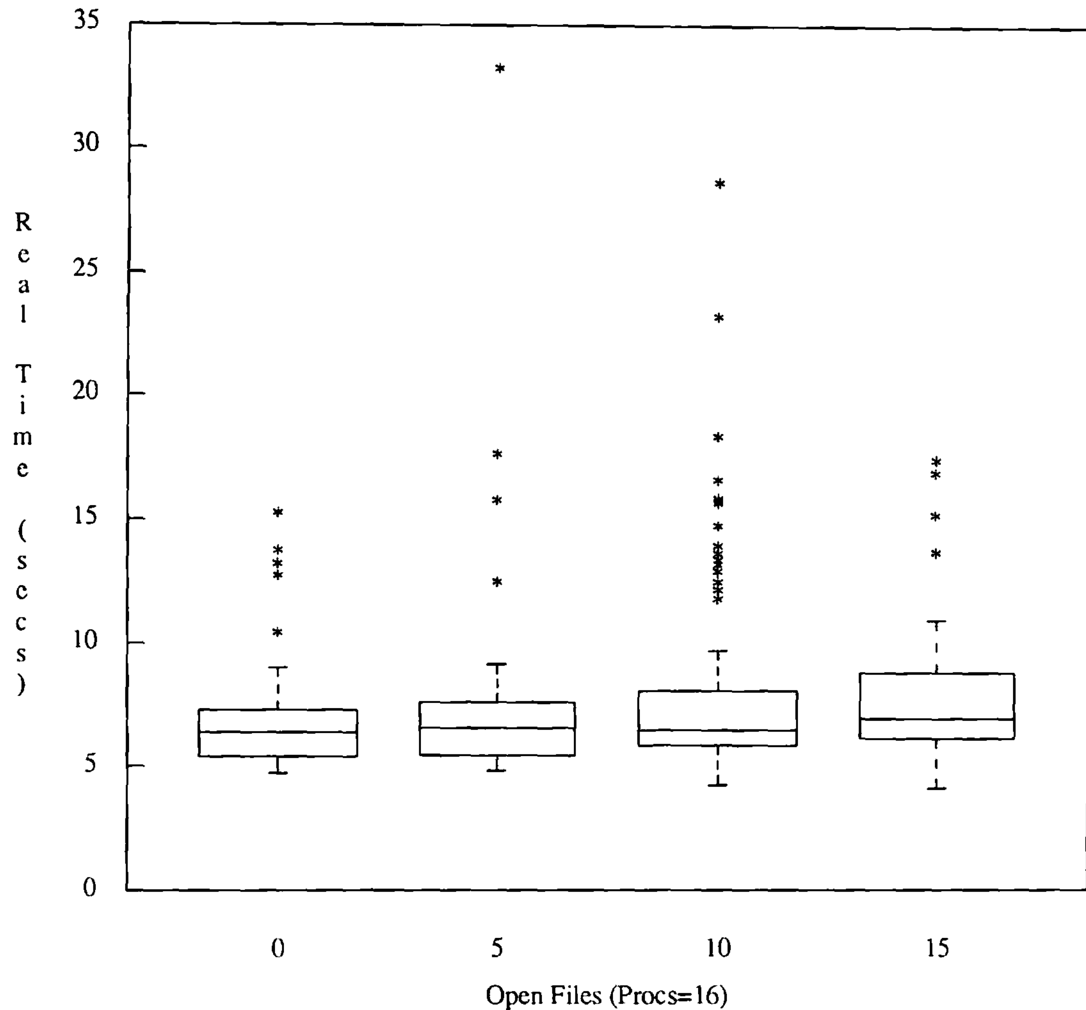


Figure 33: Effect of number of open files on real time, 16 procs
(For 100 repetitions)

Surprisingly, asynchronous elimination still does not have any effect on the real time, as illustrated in Figure 34. These results inspire a certain amount of suspicion, since the synchronous version must unfailingly *wait()* for each of 16 children to complete, while the asynchronous case (at least according to the code) returns without *wait()* -ing: collection is done outside of the timing loop.

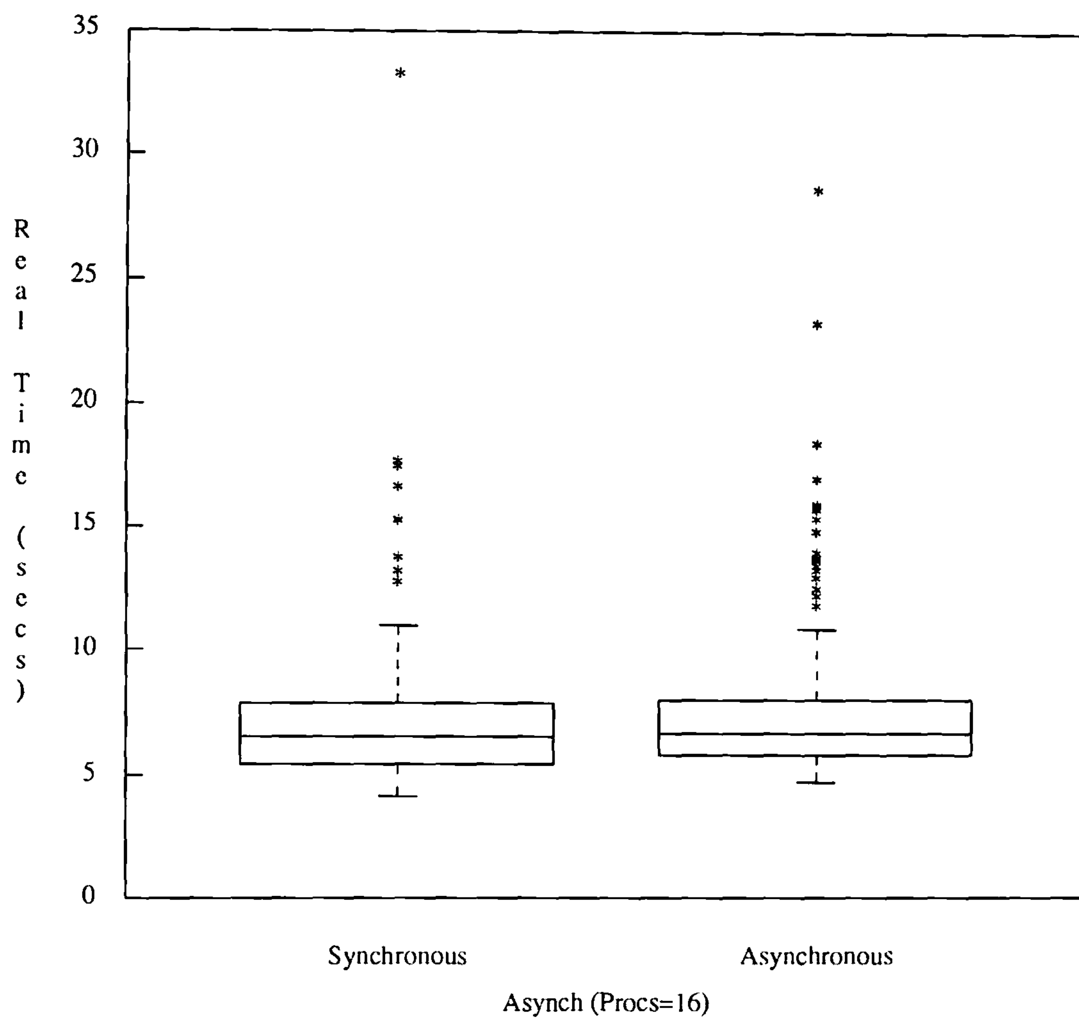


Figure 34: Effect of Asynchronous elimination on real time, 16 procs
(For 100 repetitions)

Figure 35 indicates that dirtying pages has a slight effect on the real time: we doubt that the effect is significant.

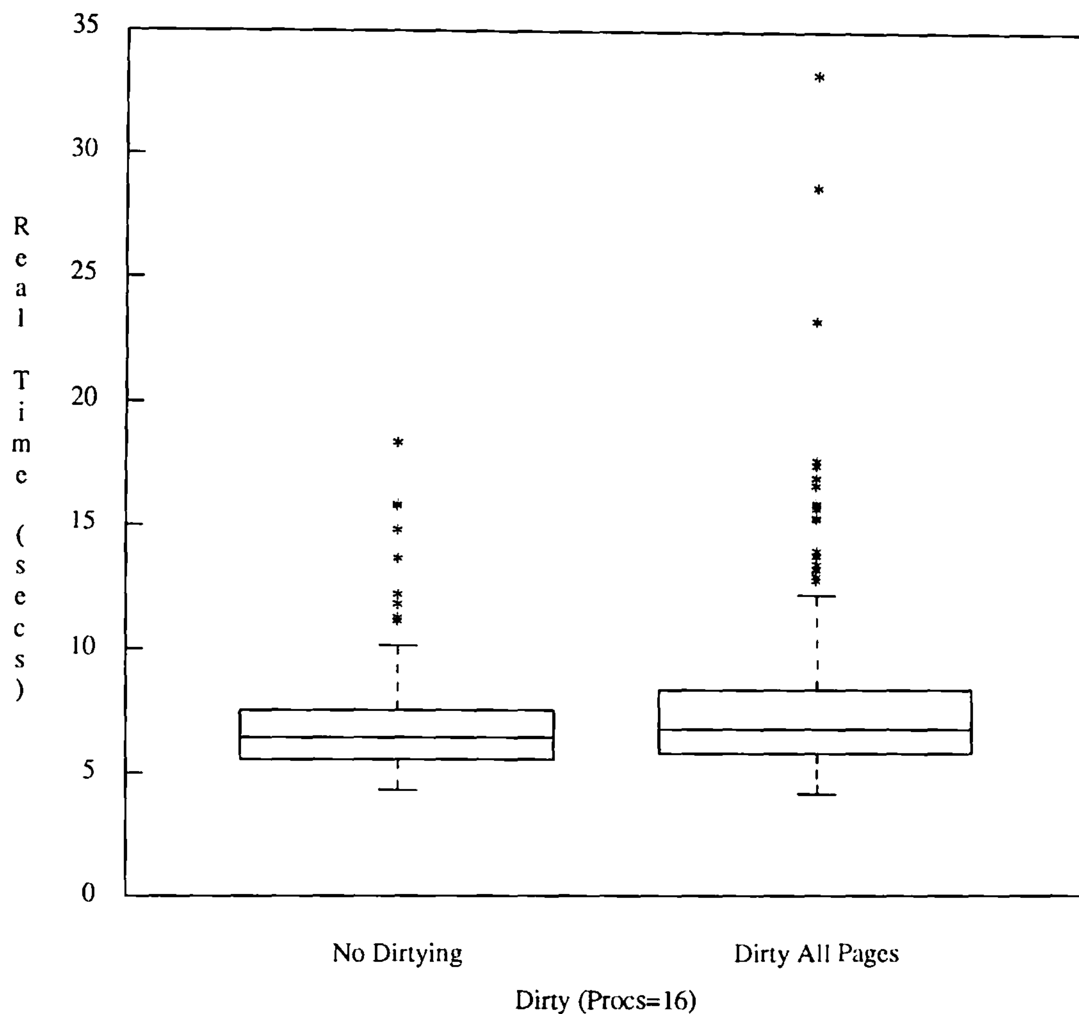


Figure 35: Effect of Dirtying child pages on real time, 16 procs
(For 100 repetitions)

Child process computational activity indicates a minor influence on real time, as shown in figure 36. Work=1, that is, the process is computing rather than sleeping, seems to cause a greater amount of dispersion as well as a slight increase in the median value; little significance can be attached to the observation.

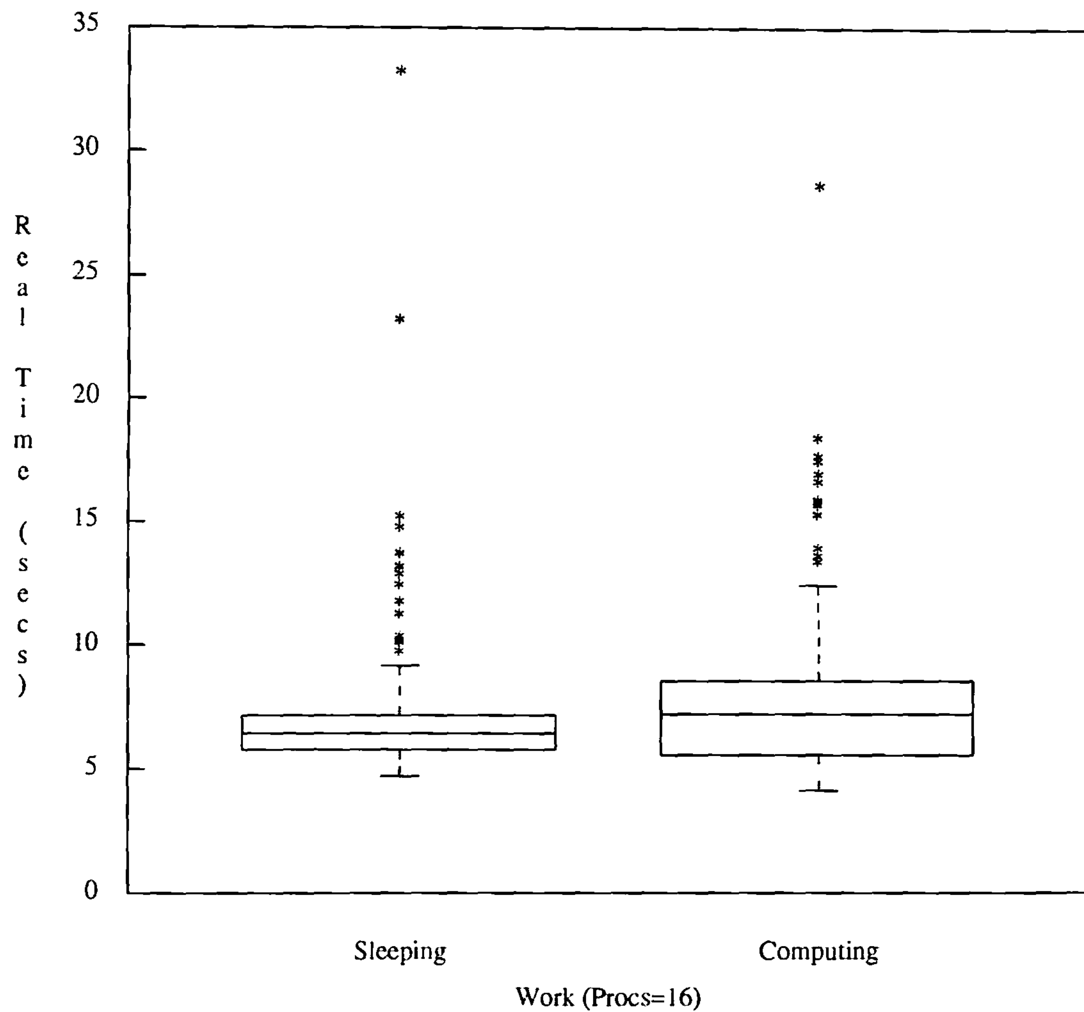


Figure 36: Effect of child work/sleep activity on real time, 16 procs
(For 100 repetitions)

Process size has a greater effect on the measured real time in the 16 process case than we observed for all the values of Proc lumped together. This effect is demonstrated by figure 37. The fact that the effect begins to be significant only on the larger sizes, not linearly or proportionally, makes us suspicious that the cause is some system artifact such as a limitation on the system memory size forcing a *swapout* of one or more of the processes under test.

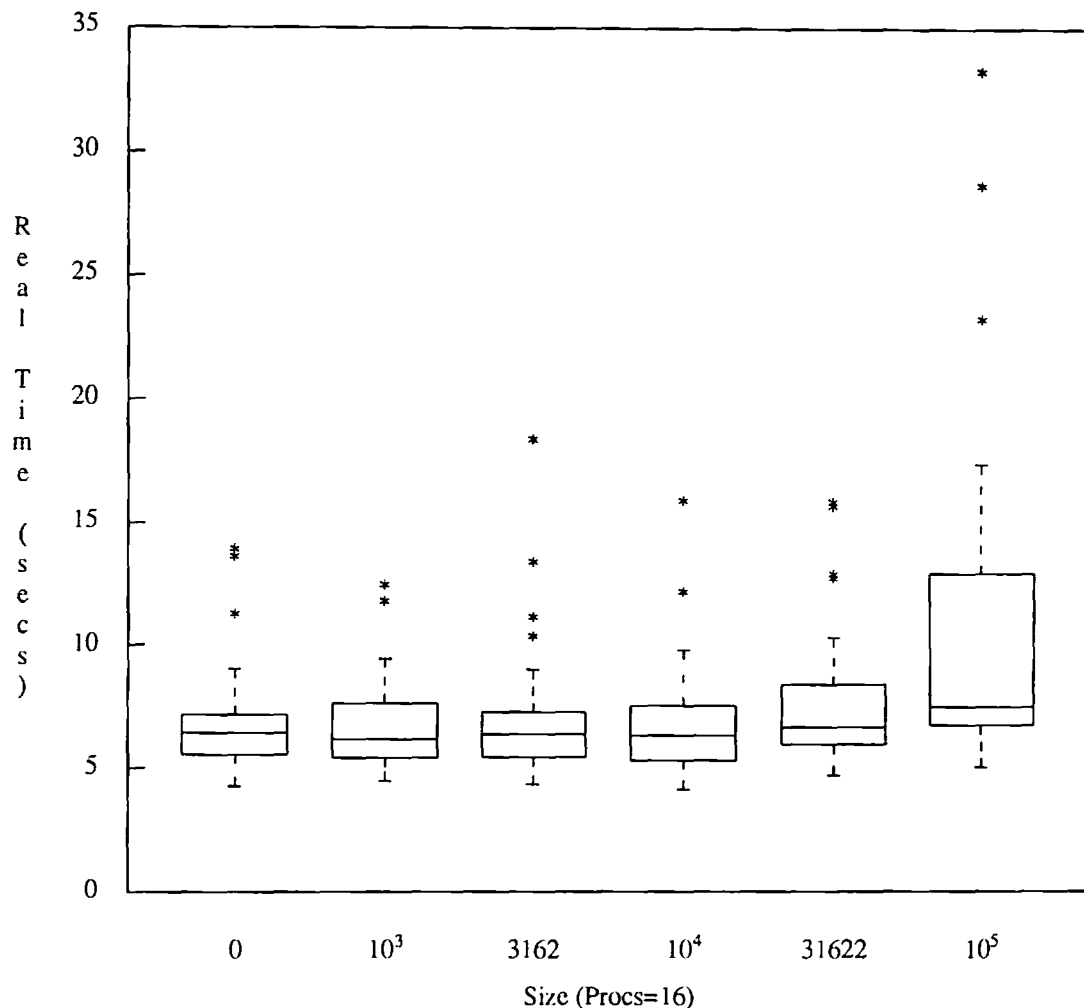


Figure 37: Effect of child process size on real time, 16 procs
(For 100 repetitions)

What we can infer from the data for the special case of 16 spawned and terminated sub-processes is that the observed influences on real time are pretty much the same. The effect of process groups is more dramatic, and there began to be an influence exerted by the size of the processes. There was also an observed, but likely to be insignificant, increase in costs associated with the number of open files at elimination time. What the 16 process case should illuminate is the increasing effects of various variables with a *change of scale*.

We also thought it provident to examine the case of 16 processors for system time as well, to isolate any influences which might not otherwise be visible. This data is

examined in the next section.

4.6.4. System Time, 16 Procs only

As before, the Group signaling mechanism seems to have a significant effect on the observed performance, in this case with respect to system time. While the real time impact is about a factor of 1.3, the system time impact is about a factor of 3; comparison with the earlier graphs shows that clearly, the effect of this variable increases with the number of child processes.

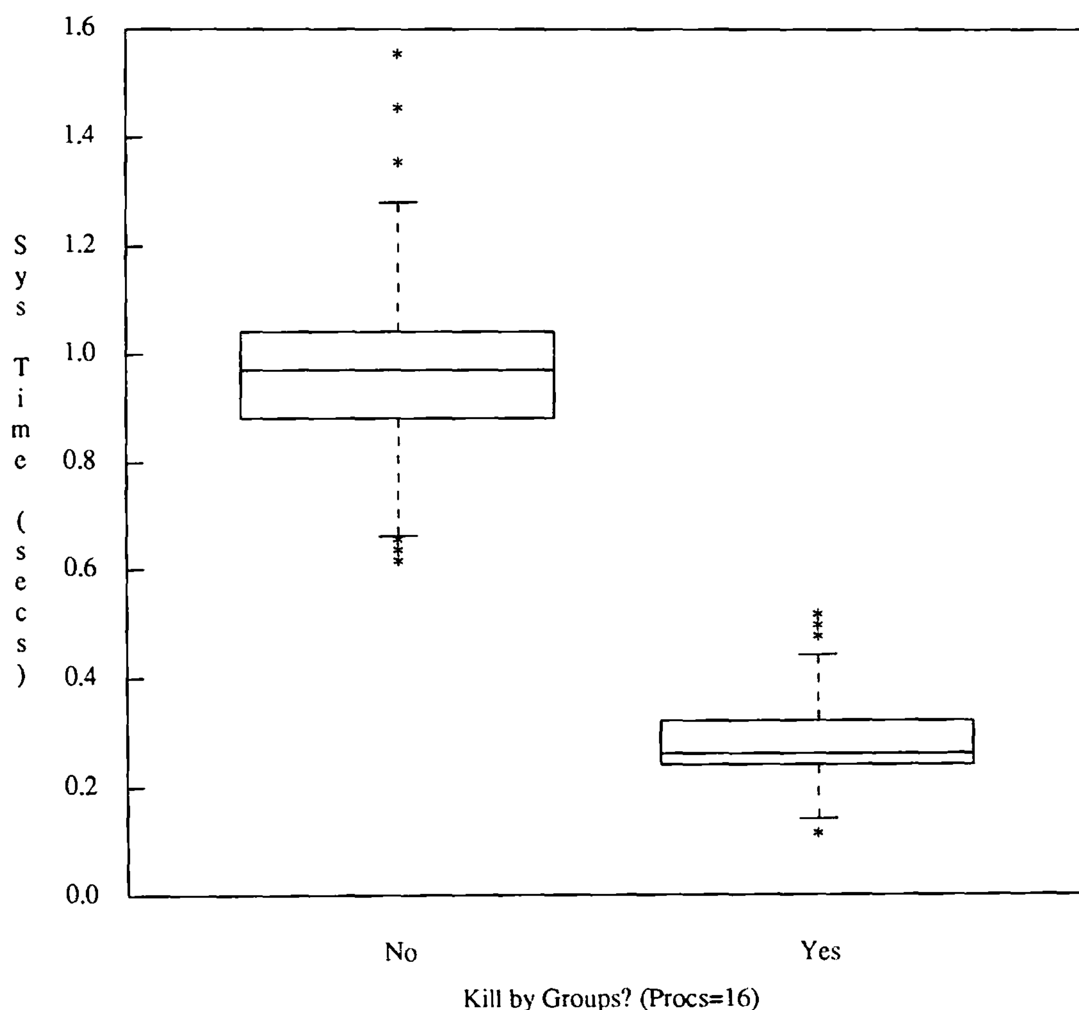


Figure 38: Effect of signaling process groups on sys time, 16 procs
(For 100 repetitions)

Open files, shown in figure 39, have little effect on the system time for sixteen processes:

this suggests that the results observed for real time with 16 processes may be spurious.

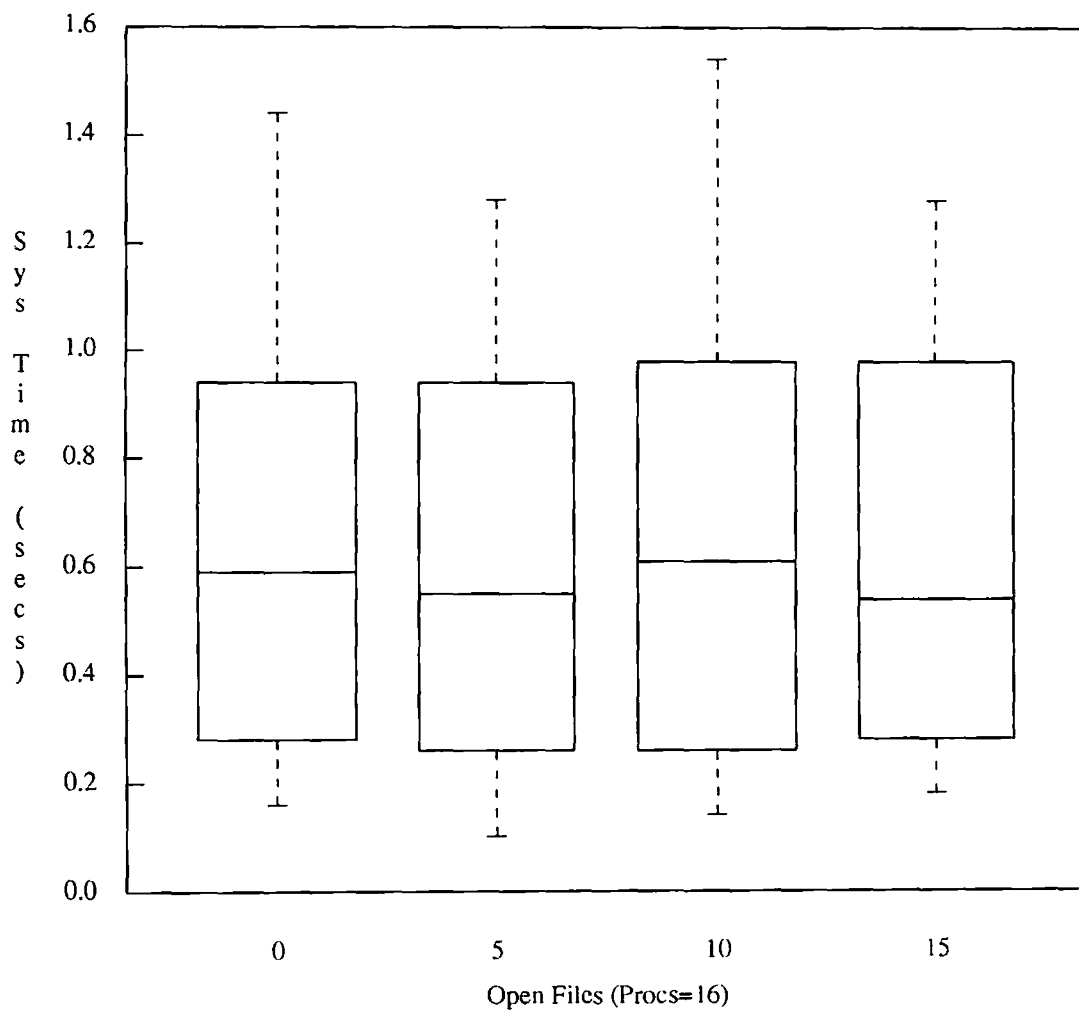


Figure 39: Effect of number of open files on system time, 16 procs
(For 100 repetitions)

Asynchronous elimination had no effect, as is illustrated in figure 40; this is interesting only in that it correlates well with the previous observations.

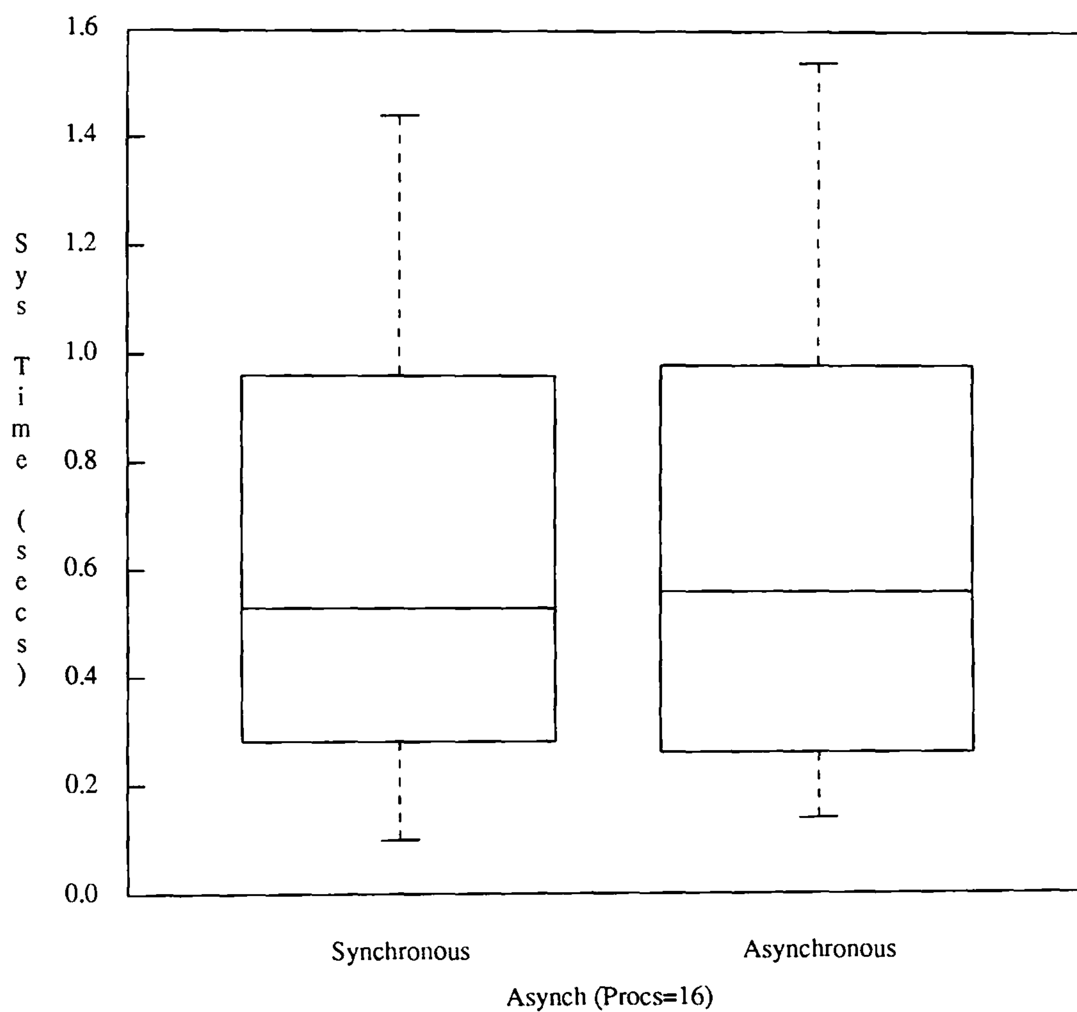


Figure 40: Effect of asynchronous elimination on system time, 16 procs
(For 100 repetitions)

Dirtying had no effect on the system time, as figure 41 illustrates.

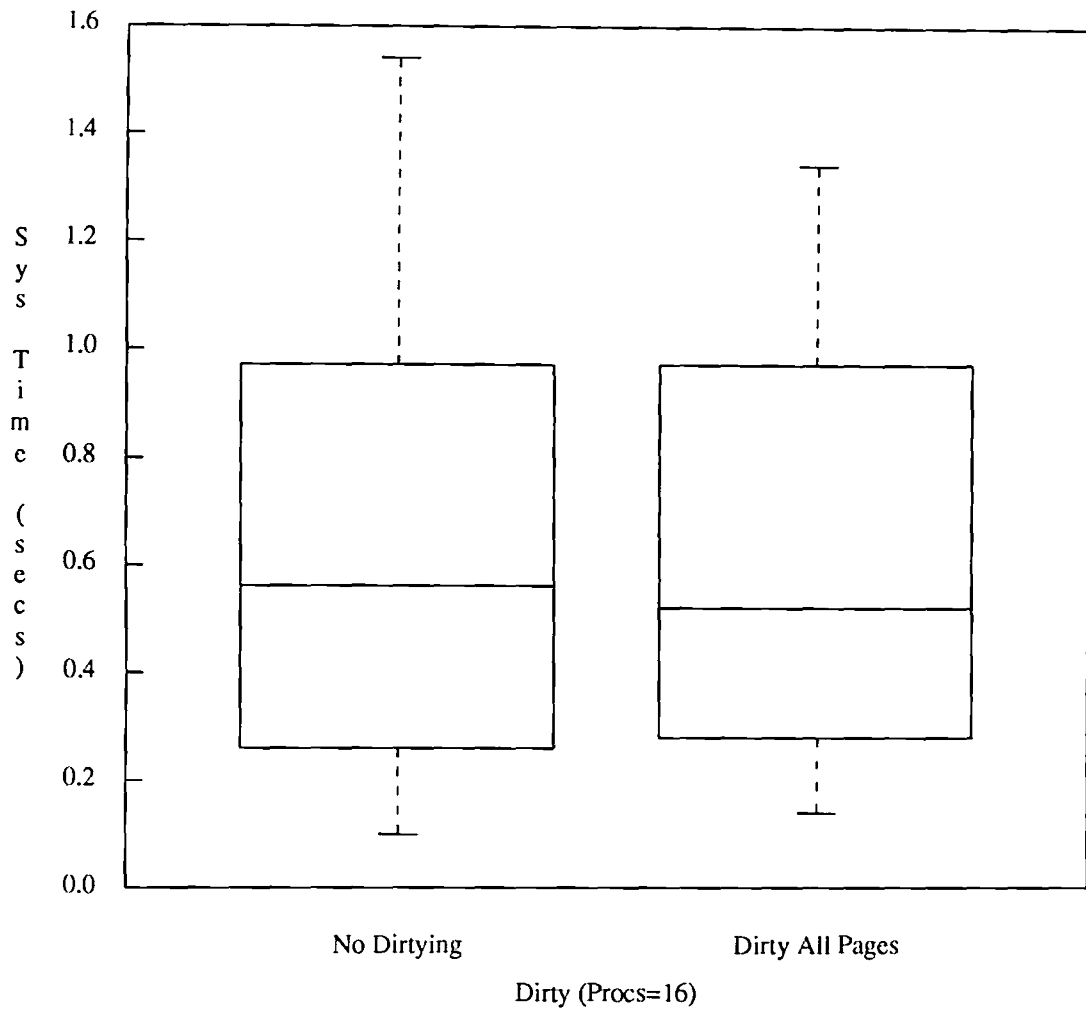


Figure 41: Effect of dirtying child pages on system time, 16 procs
(For 100 repetitions)

Processing behavior of the spawned child processes had no effect on the system time, as figure 42 illustrates.

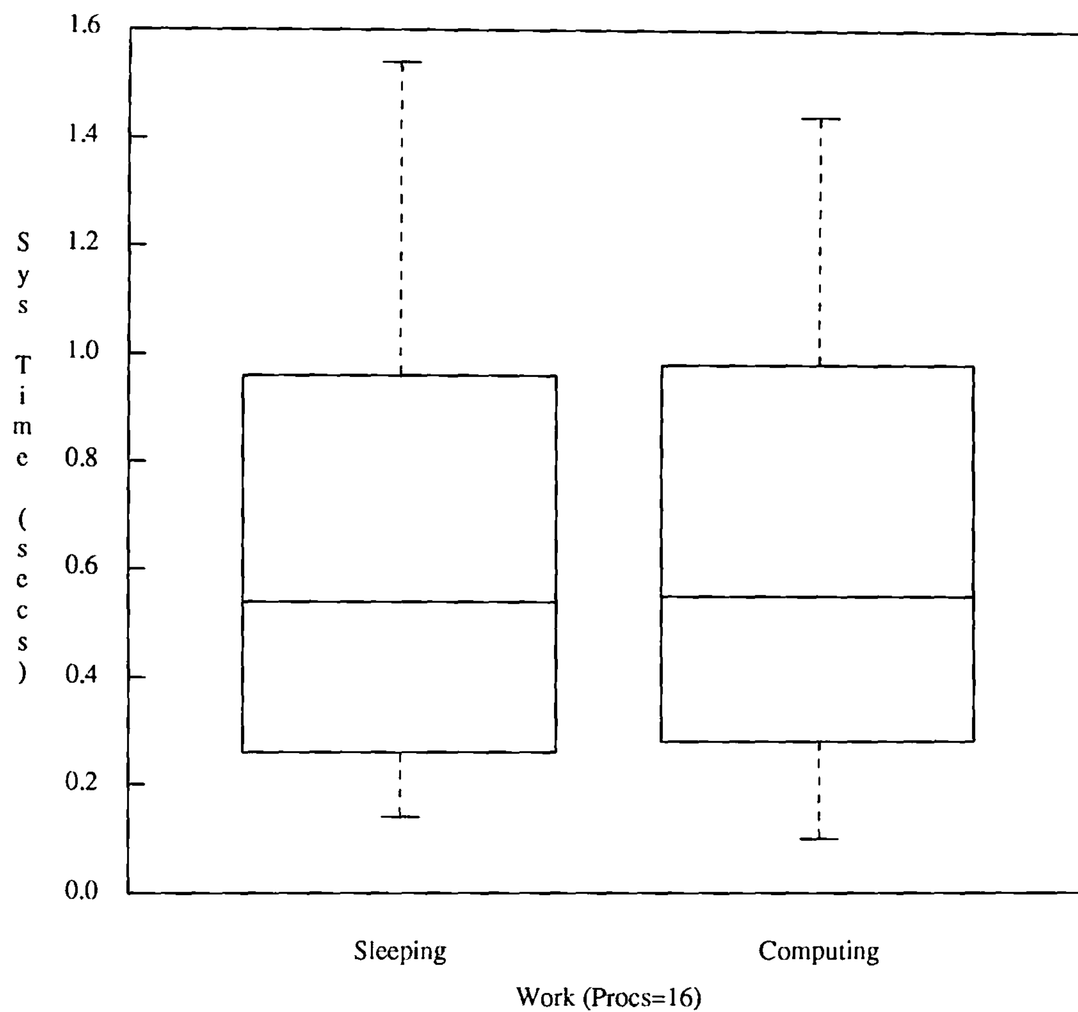


Figure 42: Effect of child work/sleep on system time, 16 procs
(For 100 repetitions)

And finally, process size had no effect on the system time, as figure 43 illustrates.

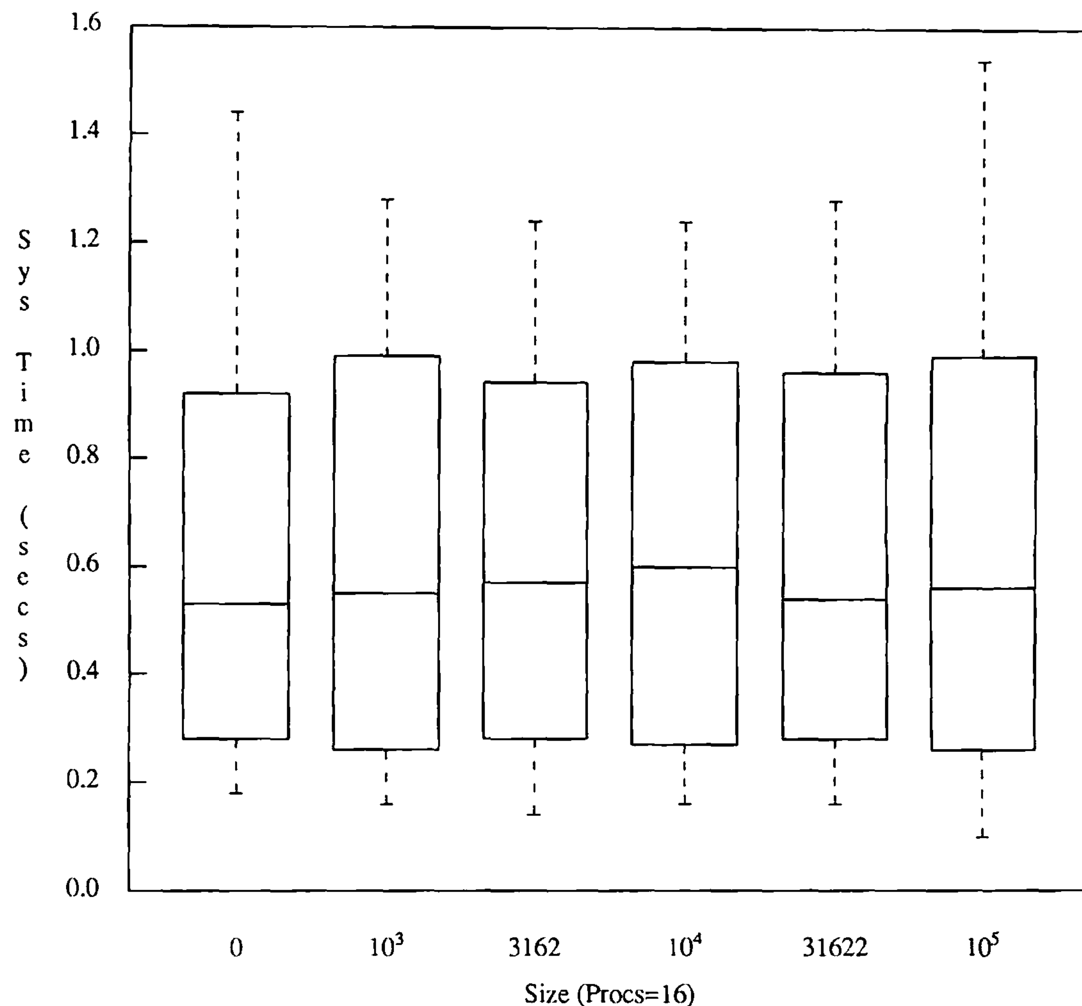


Figure 43: Effect of child process size on system time, 16 procs
(For 100 repetitions)

After reviewing the results of the last four sections, it seems clear that such factors as the number of open files, subprocess actions in executing instructions and dirtying pages, and for the most part, the sizes of the child processes have little effect on time required for sibling elimination.

The time increases with the number of processes, and this was used to see what factors change with scale; there were really no significant changes. What was surprising, however, was the major impact of process group signaling, and the non-existent impact of asynchronous execution. After some further examination of the code, we changed the code so that the *do_elim* process forked a copy of itself just previous to the sibling

elimination phase. In the synchronous case, the termination of this process was waited for. The asynchronous case returned without waiting. All children and grandchildren are eliminated by a post timing-loop cleanup routine.

This had a dramatic effect on the timing results, as is shown in the next section. We believe that the relationship between Groups and Asynch was due to a subtlety in the UNIX process scheduling which defeated the intent of our measurement apparatus.

4.6.5. Correction for process scheduling

To scale the graphs properly, all real time measurements longer than twenty seconds were removed from the graphs. This only removed outliers from the full graphs, and allows an enlargement of the relevant detail. The first graph, figure 44, of Groups against real time, shows that the effects of Groups have gone away.

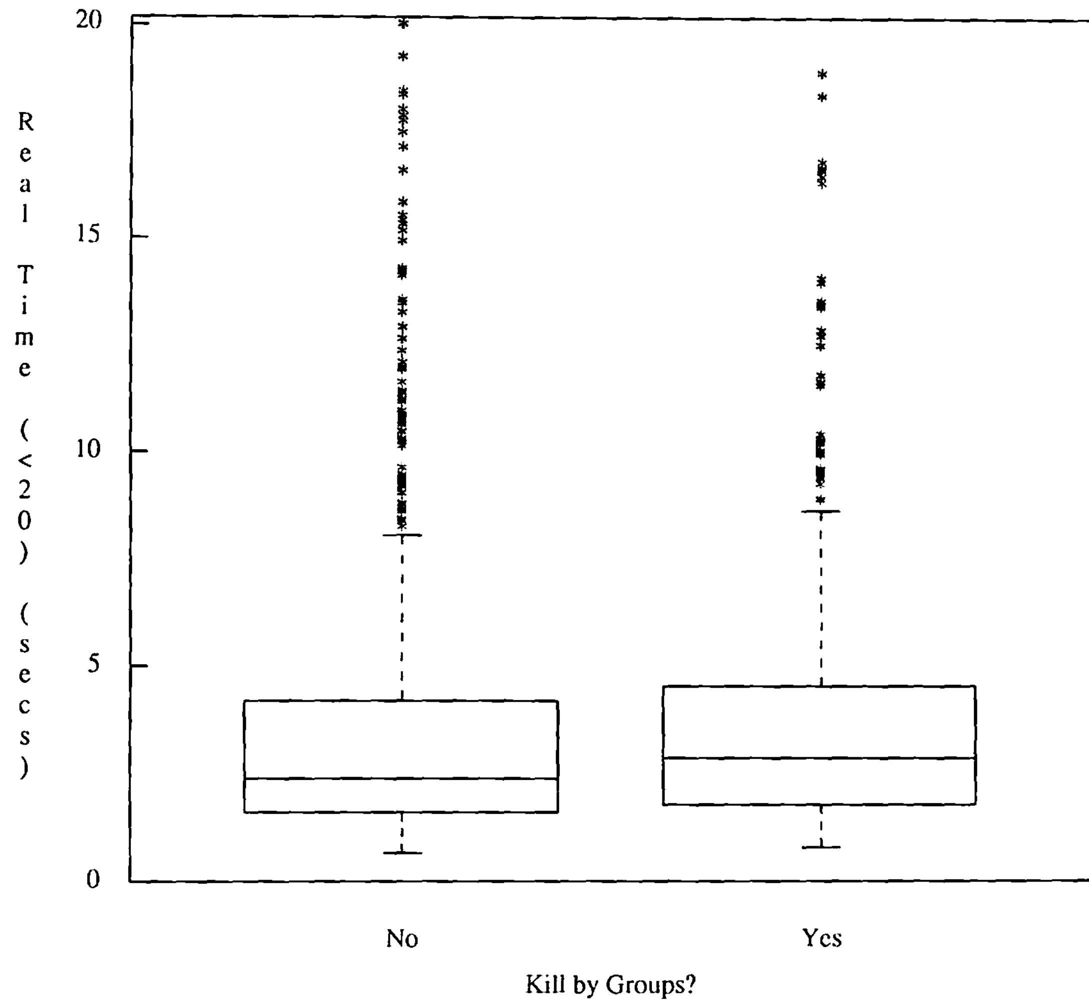


Figure 44: Effect of group signaling on real time, corrected
(For 100 repetitions)

The expected change in real time for the Asynchronous case occurred, as is shown in figure 45; note that a factor of two difference is observed.

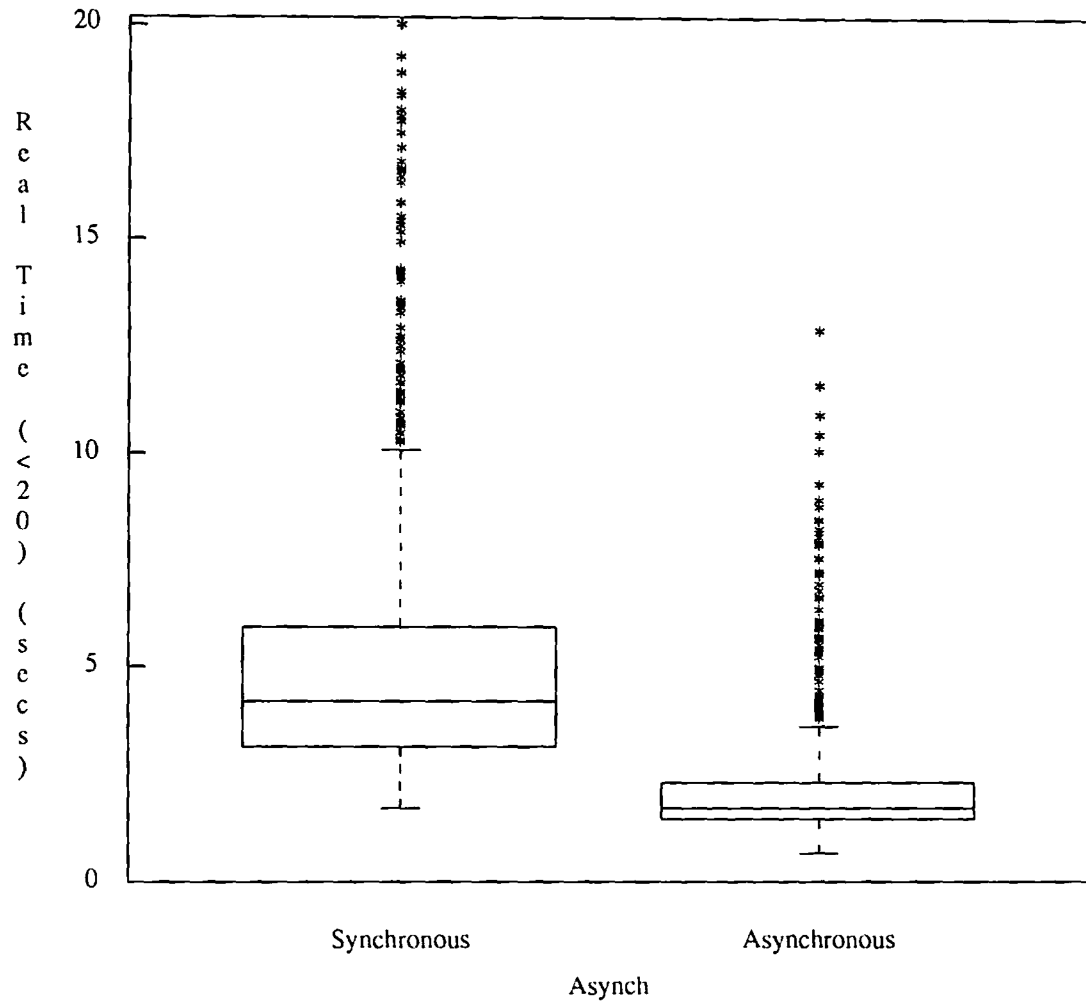


Figure 45: Effect of Asynchronous elimination, corrected
(For 100 repetitions)

Groups was examined for the case Procs=16 to see if scale changed anything. It did not, as figure 46 illustrates.

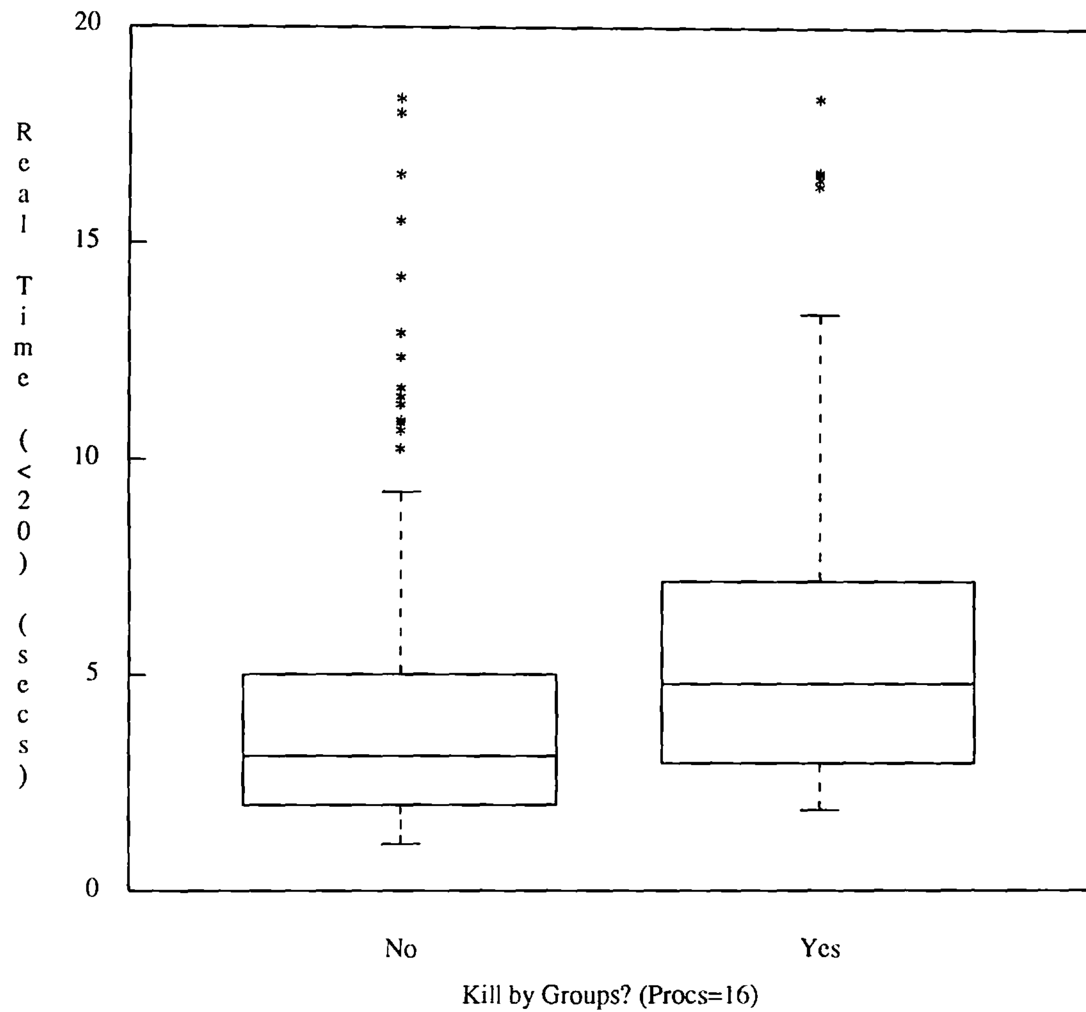


Figure 46: Effect of process groups on real time, corrected, 16 procs
(For 100 repetitions)

Finally, the effect of asynchronous elimination shows a factor of more than two improvement in the real time case with 16 processes, scaling as expected. This is illustrated by figure 47.

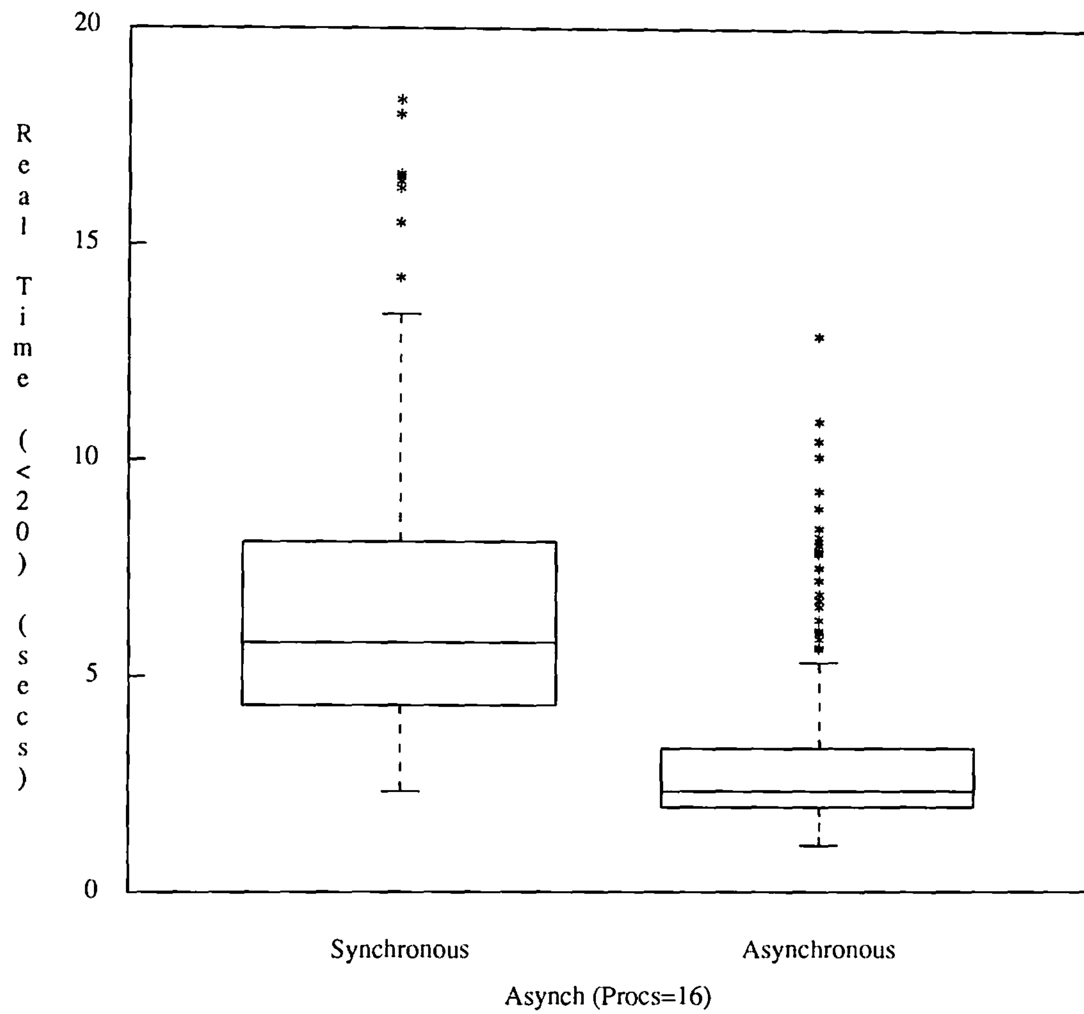


Figure 47: Effect of asynchronous elimination on real time, corrected, 16 procs (For 100 repetitions)

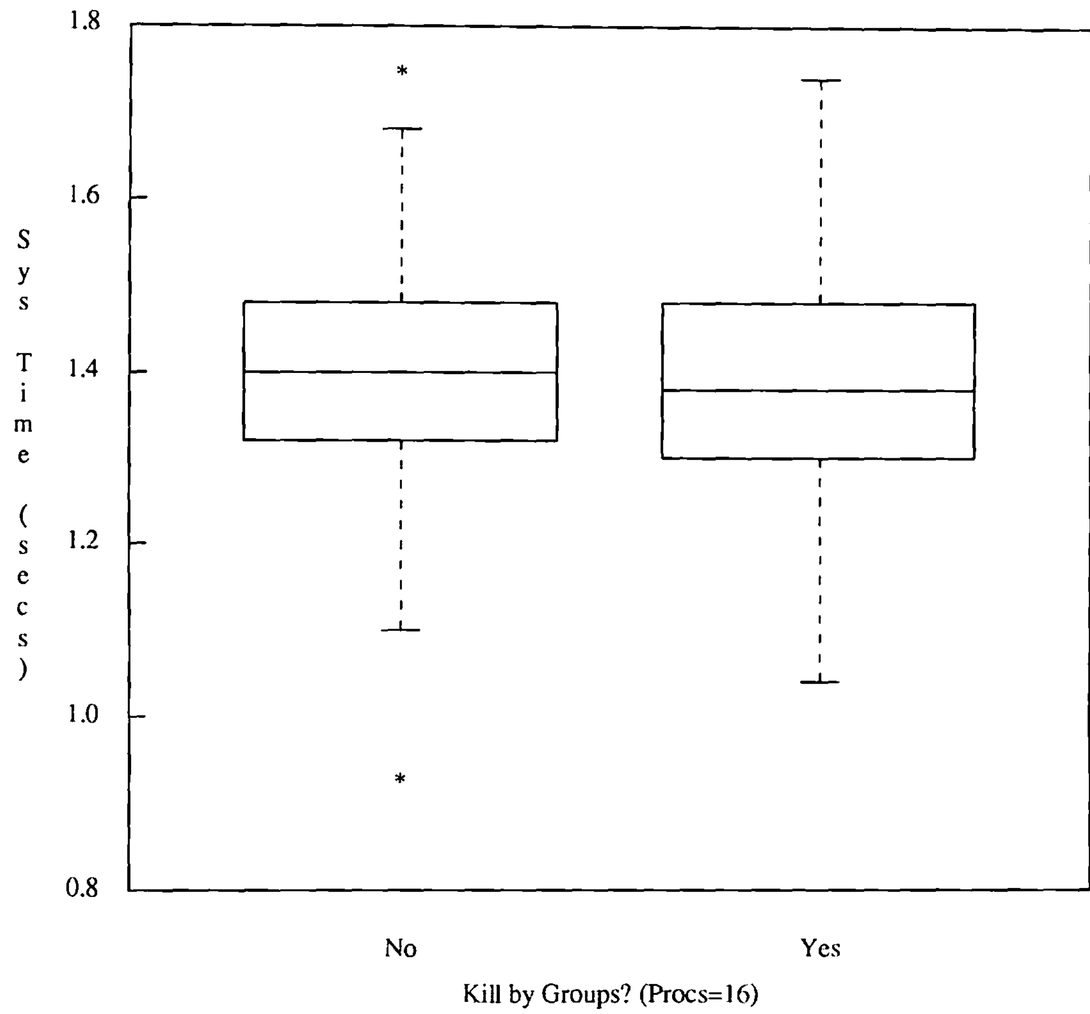


Figure 48: Effect of group elimination on system time, 16 procs
(For 100 repetitions)

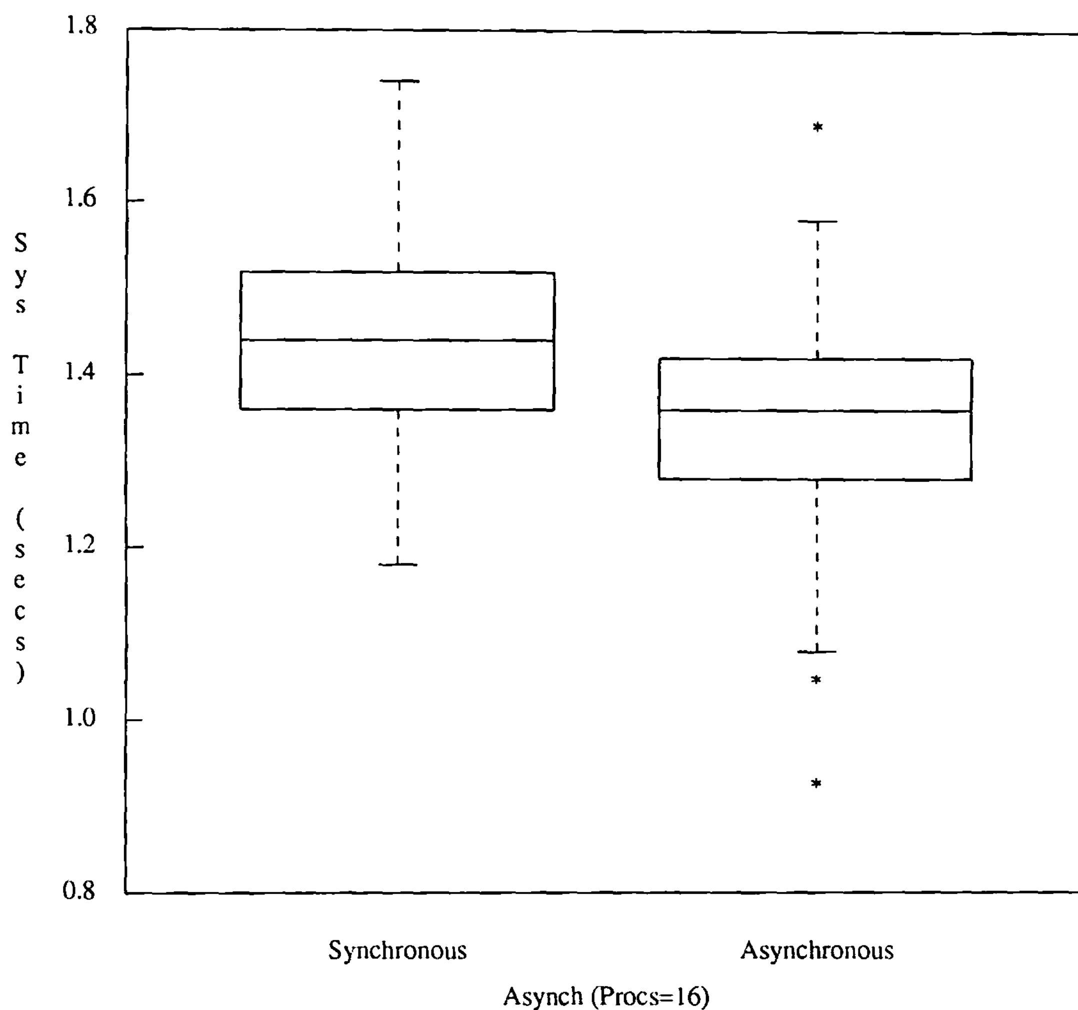


Figure 49: Effect of asynchronous elimination on system time, 16 procs (For 100 repetitions)

4.7. Possible Sources of Error in Measurements

There are two places where errors can affect the measurements. One is the measurement apparatus, in this case the UNIX timing facility. The other is in the experimental apparatus which gathers the raw data from the measurement when experiments are performed. In this discussion, the UNIX clock facility was used for all measurements, and is a common weakness or strength of the experiments.

4.7.1. UNIX Clock Facility

The UNIX clock facility is implemented [Bach1986a] as follows:

1. The underlying architecture provides a "timer" facility. The timer is typically implemented as a storage location into which a number is written. This number's value represents hardware "clock ticks'; the value is decremented by the machine clock on each of its cycles. After the hardware clock has "ticked" down to zero, a timer interrupt is generated.
2. At the interrupt vector for the clock is the address of a timer routine. The timer routine first increments a counter, the "software clock." The timer routine then examines a linked list of queued tasks (the "c-list"), decrementing the value of the first task's timer. The tasks are queued in the order they should be executed; each task's "timer" is stored as the difference between it and the preceding task on the list. Storing times this way reduces searching to update timer values; the relative value is set upon list insertion. Examples of tasks are (1) checking for characters on a teletype line and (2) sending an *alarm(1)* signal to a sleeping process.
3. The state of the system when interrupted determines which of a set of auxiliary clock timers is incremented, e.g., `USER`, `SYS`, `WAITING`, or `IDLE`. Tasks which have non-positive timer values are executed
4. The value stored in the hardware timer is determined by selecting a HZ value for the software clock, such as 60. For this value, software clock ticks are to occur every $\frac{1}{60}$ second, so if there are T hardware timer units per second, $\frac{T}{60}$ is loaded into the hardware clock

Running processes are preempted and rescheduled at the "soft" clock interrupt if they have not voluntarily released the processor. To reduce the overhead of such a timer facility, it can either be made fast, or called infrequently. The UNIX time facility was created for use on small slow machines, so that the second approach was applied after the first was inadequate. Thus, the clock granularity is large relative to the execution speeds of current processors. For example, the workstations we used for our measurements execute about 2 or 3 million instructions per second. So, in $\frac{1}{60}$ second, about 40,000 instructions can be executed, which is sufficient for several significant system events, e.g., context switches, to take place. Most system events, e.g., the execution of a system

call, require less than one "tick," so that timing a single system call will often show that the call took no time at all. Yet these events, for example spawning a new process with the *fork()* system call, are precisely what we are interested in measuring. Thus, the clock timer seems to be an insurmountable error source; the difficulties it causes in performance measurement have been noticed before [Johnson1987a]

There are two techniques which can be used to gather useful data in spite of the large error caused by the clock granularity:

1. One trial consisting of repeated experiments from which the mean value is returned.
2. Repeated trials.

The mean of repeated samples is calculated by performing a series of measurements and timing the entirety. The value is divided by the number of experiments to get the mean. The difficulty with this technique in computer system measurement strategies is that architects have constructed system features to perform faster under this behavior, e.g., caches. Thus, the measurements may not truly be indicative of the cost of performing a single operation. The "random block number" approach of *netrand.c*, given as Appendix II, attempts to circumvent these caching strategies to give a truer reading for the network page access time.

Repeating trials of the experiment indicates that the experiments are reproducible. Reproducibility, a fundamental requirement of science, reduces the possibility of some artifact such as a network overload temporarily perturbing the measurements. Such an artifact would not be present in the isolated system under study. These techniques were applied in the development of our experimental apparatus, namely the programs in Appendices I, II, XI and XII.

4.7.2. Experimental Apparatus

We discussed the effect of repetitions on the reproducibility of experiments in the discussion of "copy-on-write" fork performance. It was clear that a small number of repetitions produced data that was not reliable, as the poor qualities of the clock for fine grained timing measurements displayed themselves. Repeating the inner loop and deducing an aggregate value produces results that are *consistent with prediction*. The prediction in this case is not sophisticated; it's mainly that "copy-on-write" has the most effect when little is copied. The value of this "prediction" is that it gives us a quick check on

the results, and it's clear that small numbers of repetitions do not adhere to the model. Our study of copy-on-write fork performance gave a graphic illustration of the effect of repetition. We also demonstrated relations between different types of time measurements (which should be closely related) which change as a function of the number of repetitions.

Repeated trials test the validity of the experiment. For example, if the experiment is to repeat some action 100 times and derive a mean execution time, repetitions of this experiment should produce approximately the same mean execution time, under the assumption of a reasonable variance. If they do not, then the figure for the mean execution time has no *experimental validity*. In science, since other researchers cannot reliably reproduce the number under the same conditions, it has **no** value. All experiments reported in the measurements section were repeated at least twice in order to ensure some measure of validity. Other validation comes from repetitions with adjusted variables. Predictions coming from models of system behavior will give estimates of the *sensitivity* of the experiment to various external factors, e.g., memory size or machine speed. If, for environmental factors to which the experiments are insensitive, the experimental results appear to be duplicates, the results have validity.

To test the results of the "sibling elimination" measurements, which seemed to be only grossly reproducible, we tried another strategy. The problem with the clock timer is that its granularity is constant across systems of widely varying performance. Modern processors are sufficiently fast that events can escape detection by the timer. On a slower processor, the clock *appears* to be of finer granularity. A subset (for example, 1, 4, and 8 child processes) of the *do_elim* experiments were re-run using an AT&T 3B2/310, which as we have mentioned earlier, is significantly slower (by about a factor of 7) than the HP9000. The curves generated were consistent with the measurements we presented in this chapter. The consistency of the curves implies that the effort taken to limit the effects of errors in the measurement apparatus were successful. The validity of our conclusions is reinforced by the relationships maintained between parameters. These stay the same even as the magnitudes change due to the slower processing unit.

4.8. Discussion

We have discussed use of alternatives and shown how execution of the alternatives in a concurrent fashion can lead to performance increases. Further, we have parameterized characteristics of the alternatives and the implementation overhead so that the effectiveness of the technique can be evaluated. Many of the contributors to $\tau(overhead)$ have been identified and measured, so that the performance improvement can be evaluated on the basis of the alternative's characteristics, using these characteristics to calculate $\tau(overhead)$.

4.9. Applications

> *''In science the primary duty of ideas is to be useful and interesting even more than to be 'true' .''* [Trotter1941a]

What properties must we have, other than minimal implementation overhead, for the concurrent execution method we describe to be useful? The analysis of Chapter 2 and the measurements earlier in this chapter suggest several:

1. A large portion of the shared state is read-only.
2. There is some state shared between the alternatives which each may update. With no shared state, no work is necessary for transparent concurrent execution.
3. There are execution time differences between the alternatives, due to data dependencies, use of heuristic methods, or other influences.

Application areas for our design are described in the following sections.

4.9.1. Distributed Execution of Recovery Blocks

The Recovery Block [Horning1974a, Randell1975a] is a method for writing software which is tolerant of mistakes in its own logic, from which failures can arise. The idea is simple. It is assumed that the software in question has been written to some specification. Several alternative versions of the software are written, according to the specification. A boolean ''acceptance test,'' which checks the results of the software is developed along with the software, using the specification. The acceptance test, which either succeeds or fails, will be refined once some experience with the software is developed.

The alternatives and the acceptance test are gathered into an ALGOL-like block

construct, where the alternatives are typically ordered on the basis of observed or estimated characteristics such as reliability and execution speed.

When the acceptance test succeeds, the results (including all state changes) of the alternative which passed the test are made available. When the acceptance test fails, the state of the program is ''rolled back'' to the state the program had before the block was entered, and the next alternative is tried. If the last alternative in the sequence results in a failed acceptance test, the block as a whole fails.

The scheme is conceptually similar to the "standby spare" technique used in hardware. *N* alternate methods of passing an *acceptance test* are provided. The first such method is referred to as the *primary;* they have typically been rank-ordered by some metric, e.g., observed performance. Assuming that the acceptance test performs perfectly, the recovery block method fails on inputs where all methods fail the acceptance test. Note that the acceptance test is application-specific; Hecht [Hecht1979a] provides a detailed discussion of the forms such acceptance tests might take. Cha, *et al.* [Cha1987a] have shown that self-checks (a generalization of the acceptance tests used by recovery blocks) can be effective in finding faults. However, there is difficulty both in the writing of the self-checks and their placement within the program structure. They also note a great variation in the ability to write effective self-checks, and the efficacy of combining code-based checks with specification-based checks compared to specification-based alone.

The recovery block is different in behavior than the ''Alternative Block'' we proposed as a sequential model in the Introduction. First, rather than having one guard per body, the Recovery Block possesses one guard to which all the alternatives are passed. Second, the guard is applied *after* the body is executed, rather than before. However, neither of these are problems for our design, as (1) the computation can be viewed as part of the guard, with the body consisting solely of updates to external variables, or (2) the blocks can be viewed as self-checking entities where the guard is always enabled for scheduling of the computation, but which may fail due to self-checks.

The changes to the program's state space are equivalent to some execution which selected exactly one alternative at each Recovery Block encountered in the execution. This the nondeterministic selection which we discussed in the introduction. The RB language, developed by Smith and Maguire [Smith1988c] is designed for the distributed

execution of recovery blocks.

Since Recovery Block alternates may attempt to update shared state, e.g., database files or external variables, our mechanism for preventing observation of a sibling's actions is necessary, and the "copy-on-write" memory management reduces the amount of state which must be maintained. One special problem which arises with the parallel execution of Recovery Block alternates is the fact that the method is designed to cope with failures, so that we must do more work in order not to add new failure modes. Two issues in particular are important. First, we may copy all the state rather than copying as necessary, in order that the state not become inaccessible[14] and so cause a failure. Second, the synchronization must not introduce a single point of failure. This is remedied by the use of majority consensus, as discussed above, to achieve a fault-tolerant 0-1 semaphore for use in synchronization. A combination of existing techniques [Schneider1983a, Schneider1982a] may be needed for the distributed case.

4.9.2. Polynomial Root-finding

The Jenkins-Traub [Jenkins1972a, Jenkins1970a] algorithm for finding roots of polynomial equations is a highly robust, rapid, and portable method for solving for the zeros of a polynomial over the complex numbers. Those interested in the internal workings of the algorithm should consult Ralston [Ralston1978a] for details. The property of the algorithm which makes it attractive as an application for our method is that it is *adaptive*, in the following sense: as it begins the search for a zero in the complex plane, the algorithm chooses an angle with which it attempts to approach the root. After iterating towards the root, the algorithm tests for convergence. If the iteration is not converging, the algorithm retries with another angle, which in the published version (ACM Algorithm 419: "*Zeros of a Complex Polynomial*") is a fixed rotation in the plane. The application of the method for concurrent execution is in the choice of the angle of approach. Several such angles are selected, and alternative root-finders are spawned with the angle set. If an

[14] Evidence [Gray1988a] from commercial systems [Bartlett1978a, Borg1983a, Bartlett1981a] indicates that reliability is increased significantly with two copies of the state, as maintained in a "process pair." The increase in reliability by adding a third process is infinitesimal, and is achieved at a significant cost in storage and synchronization. Thus, if the location of storage copies is well-known, it may not be necessary to have more than two copies in the system; demand-copying can be used for any copies beyond the two necessary for reliability.

alternative algorithm does not converge, it "fails" by exiting; successful alternatives are selected using our "fastest first" synchronization scheme; our performance analysis of the speedup applies here as well. The minimum time is the CPU time it took for the first rootfinder to discover all the zeros of the polynomial. Failures are excluded, since they did not discover all the zeros. The maximum time is the CPU time required for the last rootfinder to discover all the zeros of the polynomial, or fail. Failures are included, since we would have to wait for them if we didn't know whether the rootfinder was converging. The average time is the mean of all execution times, including failures. These times are determined on a uniprocessor, since each angle choice will be executing on such a machine in the experimental work.

The granularity of finding a single zero may be too fine to dominate the overhead involved, but alternatives can easily be spawned which retry using different rotations or different starting angles for the entire set of zeros in this case.

As a test, we re-coded the published Fortran program for the Jenkins-Traub method in "C." This is provided as Appendix XIII. A variety of test cases were run against the algorithm, using a driver which varied the initial angle for the sequence of shifts. The published algorithm had used 94 degrees as an angle; we parameterized *cpoly()* with an `angle` argument which allowed different choices to be tried. The driver program, *cvaryangle.c,* is given as Appendix XIV. The compiled program's address space had a write fraction of less than 0.25 due to the large portion comprised of program text.

The test cases were generated using *r2p.c* which is given as Appendix XV. The idea is to generate a distribution of roots, which are used as input to *r2p*. *r2p* constructs a monic polynomial having these roots, and outputs its coefficients into a form usable by *cvaryangle.c*.

4.9.2.1. Example

Consider the following set of points, which are to be the roots of a polynomial:

-1.1 + 0.1i

-1.1 + 0.1i

-1.0 + 3.1i

-1.2 + 0.334i

-1.3 + 1.276576i

-0.4 + 0.026i

1.5 + 1.1i

0.3 + 1.0i

0.4 + 1.0i

0.5 + 1.0i

-1.6 + 0.0i

-0.7 + 0.76i

-1.8 + 1.9i

-0.9 + 0.0i

0.1 + 0.042i

0.1 + 0.042i

-1.0 + 3.1i

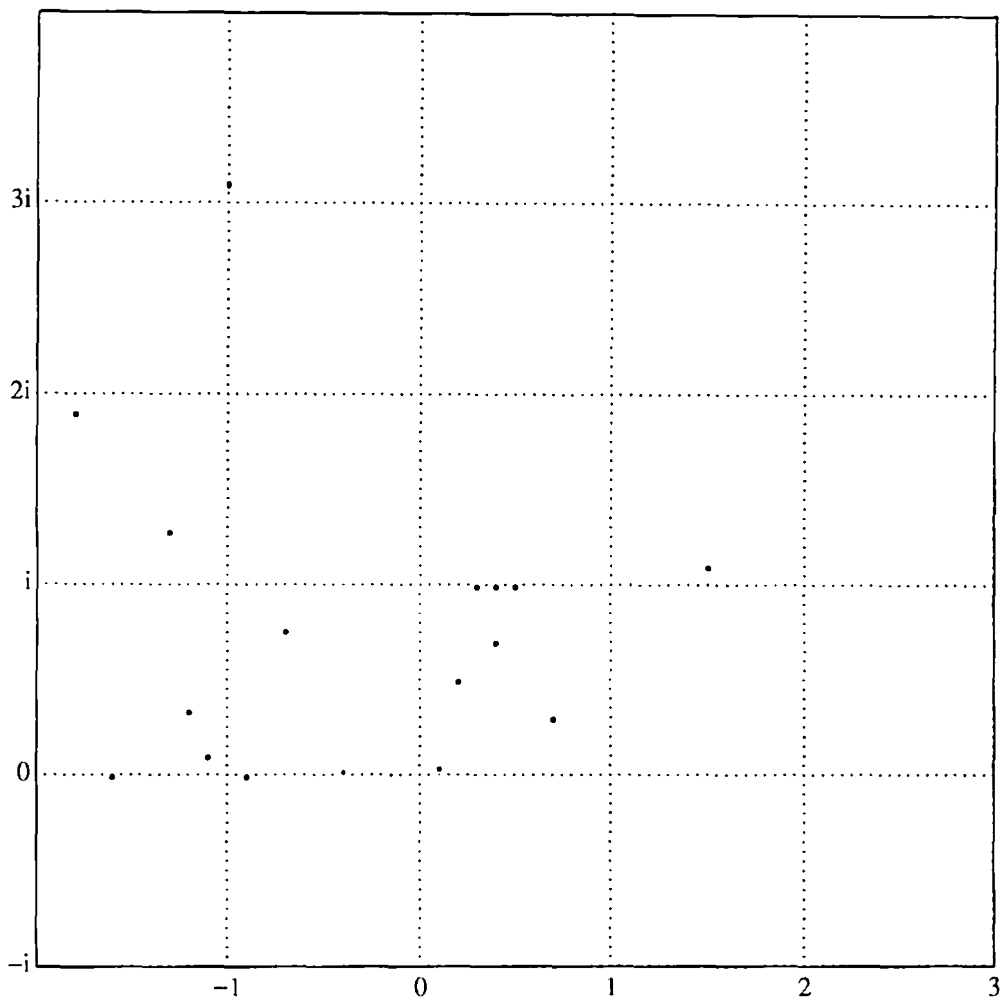
-1.0 + 3.111i

0.2 + 0.5i

0.7 + 0.3i

0.4 + 0.7i

The roots are dispersed through the top half of the complex plane:



When processed by *r2p*, they yield[15] the monic 21st degree complex polynomial $z^{21}+$ $(8.9-19.491576i)\cdot z^{20}+$ $(-140.045104-161.369678i)\cdot z^{19}+$ $(-1274.642064+381.680467i)\cdot z^{18}+$ $(-743.821643+5810.837248i)\cdot z^{17}+$ $(16752.870876+10043.831986i)\cdot z^{16}+$ $(41763.432806-29981.366628i)\cdot z^{15}+$ $(-23799.211445-106982.985377i)\cdot z^{14}+$ $(-189267.174496-34407.932181i)\cdot z^{13}+$ $(-152123.766242+236186.810121i)\cdot z^{12}+$ $(198013.830239+280742.352179i)\cdot z^{11}+$ $(342742.413912-83786.997252i)\cdot z^{10}+$ $(37460.901644-301901.877085i)\cdot z^{9}+$ $(-195080.181222-100059.575279i)\cdot z^{8}+$ $(-94732.08529+90223.058904i)\cdot z^{7}+$

[15] R2p.c has lost much precision; the real coefficients of the polynomial are different from those it computes. This approximation is not bad, P(-0.9+0.0i) is 0.8E-7, which is not far from zero. The polynomial P() for which P(-0.9+0.0i) is -0.26E-26 has vastly different coefficients. For example the coefficient of z^0 is -1088.35+607.0i [Chang1989a]. However, this precision is not necessary to prove our point about parallel processing!

$(27282.343436+57440.498715i)\cdot z^6+$ $(24271.198674-3724.963179i)\cdot z^5+$ $(682.559499-7237.864131i)\cdot z^4+$ $(-1528.494152-397.912167i)\cdot z^3+$ $(-63.349391+237.887291i)\cdot z^2+(27.878345+0.897986i)\cdot z^1+(-0.80232-1.101871i)$.

When the program of Appendix XIV, *cvaryangle.c,* is run on this input, the following output is produced on a Hewlett-Packard HP9000/350 with 8 megabytes of main memory and an HP7945 70 megabyte hard disk, running HP-UX 6.0:

```
nprocs: 1, max: 1044, min: 0, avg: 1044, fails: 1
nprocs: 2, max: 1015, min: 473, avg: 744, fails: 1
nprocs: 3, max: 1733, min: 1733, avg: 1246, fails: 2
nprocs: 4, max: 1604, min: 474, avg: 1047, fails: 1
nprocs: 5, max: 1531, min: 814, avg: 1117, fails: 3
nprocs: 6, max: 1734, min: 473, avg: 938, fails: 2
nprocs: 7, max: 1330, min: 791, avg: 1023, fails: 2
nprocs: 8, max: 1571, min: 461, avg: 1037, fails: 1
nprocs: 9, max: 1706, min: 1080, avg: 1206, fails: 3
nprocs: 10, max: 1877, min: 459, avg: 1076, fails: 3
nprocs: 11, max: 1896, min: 965, avg: 1324, fails: 1
nprocs: 12, max: 1758, min: 472, avg: 1127, fails: 3
nprocs: 13, max: 3307, min: 775, avg: 1437, fails: 4
nprocs: 14, max: 1885, min: 790, avg: 1275, fails: 5
nprocs: 15, max: 1748, min: 293, avg: 1053, fails: 6
nprocs: 16, max: 2312, min: 476, avg: 1194, fails: 2
nprocs: 17, max: 1937, min: 656, avg: 1159, fails: 5
nprocs: 18, max: 1816, min: 504, avg: 1066, fails: 4
nprocs: 19, max: 2085, min: 738, avg: 1223, fails: 2
nprocs: 20, max: 2381, min: 474, avg: 1238, fails: 3
nprocs: 21, max: 2100, min: 619, avg: 1235, fails: 8
nprocs: 22, max: 1887, min: 380, avg: 1135, fails: 4
nprocs: 23, max: 2088, min: 643, avg: 1344, fails: 3
nprocs: 24, max: 2056, min: 472, avg: 1147, fails: 4
nprocs: 25, max: 2317, min: 699, avg: 1154, fails: 8
nprocs: 26, max: 2003, min: 566, avg: 1249, fails: 6
nprocs: 27, max: 1902, min: 590, avg: 1167, fails: 5
nprocs: 28, max: 1930, min: 545, avg: 1149, fails: 8
nprocs: 29, max: 2346, min: 332, avg: 1297, fails: 7
nprocs: 30, max: 1948, min: 312, avg: 1110, fails: 6
nprocs: 31, max: 1950, min: 657, avg: 1201, fails: 5
nprocs: 32, max: 2273, min: 471, avg: 1183, fails: 7
nprocs: 33, max: 2529, min: 505, avg: 1227, fails: 5
nprocs: 34, max: 2378, min: 490, avg: 1330, fails: 4
nprocs: 35, max: 1774, min: 420, avg: 1104, fails: 7
nprocs: 36, max: 1758, min: 471, avg: 1088, fails: 7
nprocs: 37, max: 2245, min: 299, avg: 1224, fails: 8
nprocs: 38, max: 2556, min: 568, avg: 1222, fails: 7
nprocs: 39, max: 1867, min: 680, avg: 1140, fails: 4
nprocs: 40, max: 2317, min: 473, avg: 1255, fails: 5
nprocs: 41, max: 2165, min: 552, avg: 1252, fails: 11
nprocs: 42, max: 2703, min: 621, avg: 1382, fails: 7
nprocs: 43, max: 2232, min: 515, avg: 1268, fails: 13
nprocs: 44, max: 2140, min: 425, avg: 1199, fails: 4
nprocs: 45, max: 2171, min: 312, avg: 1225, fails: 8
nprocs: 46, max: 1911, min: 429, avg: 1159, fails: 7
nprocs: 47, max: 2511, min: 439, avg: 1265, fails: 5
nprocs: 48, max: 2271, min: 471, avg: 1229, fails: 9
nprocs: 49, max: 2698, min: 476, avg: 1211, fails: 6
nprocs: 50, max: 3216, min: 543, avg: 1231, fails: 9
```

The timing numbers are given in units of clock ticks; there are 60 clock ticks per second

on this machine. The relations are graphed in figure 50:

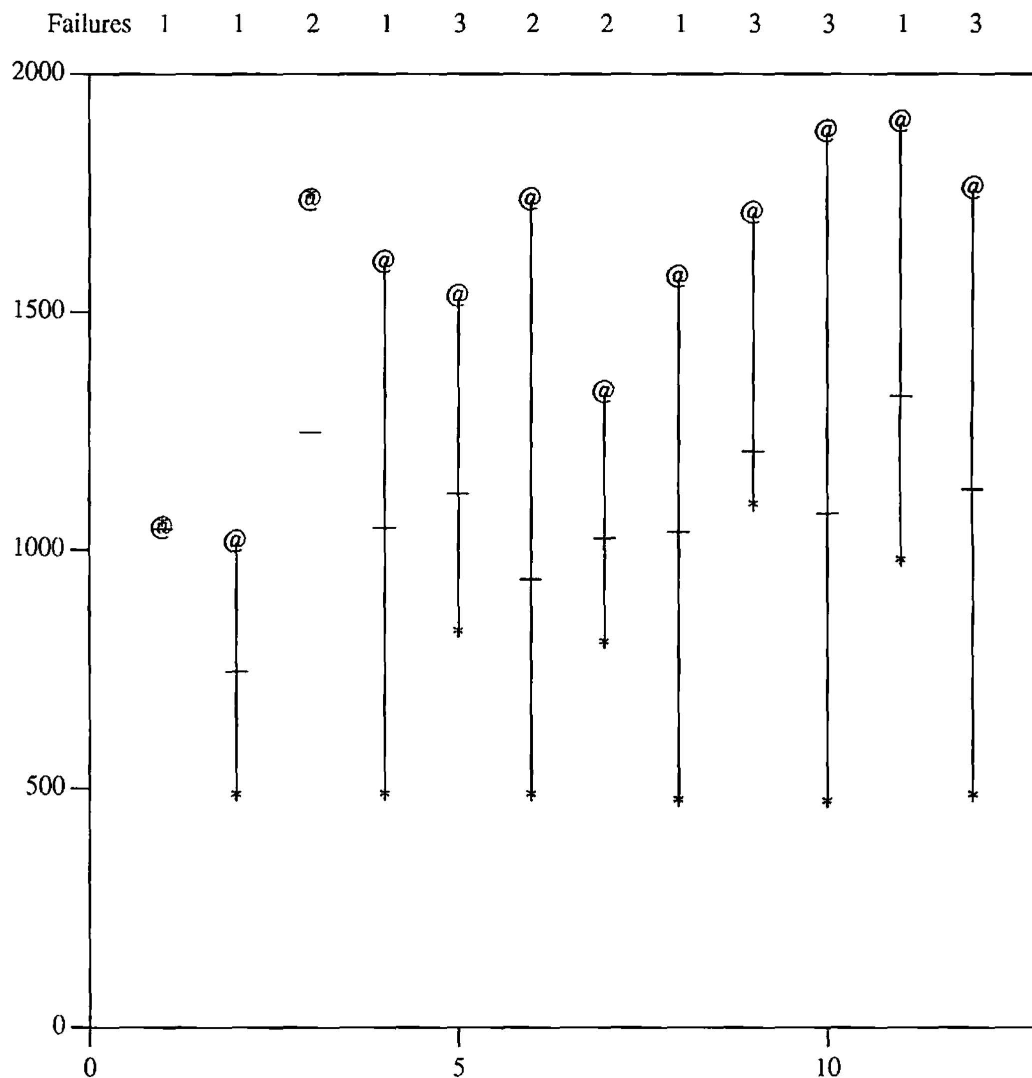


Figure 50: Varying angle versus execution time

Legend:

"*" - Minimum execution time

"@" - Maximum execution time

‘‘—’’ - Average Execution time

"Failures" - Number of choices causing failure

Times are in CPU clock tick units.

For a number of processors N, the ensemble of angles $<3.0+\frac{0\cdot 360}{N}, 3.0+\frac{1\cdot 360}{N}, ..., 3.0+\frac{(N-1)\cdot 360}{N}>$ is tried in parallel. The time for failures

is included in the averages, but the minimum time is always derived from successful executions. The number of failures is calculated by using a counter, which is output. These results illustrate two important facts:

1. A speedup is possible using only a few variations. For example, comparing the average case with 4 processors to the best case gives a speedup of about 2.2. The speedup in the two-processor case is 1.6. Of course, these measurements ignore overhead. More importantly from the point of view of a numerical analyst, the scheme tolerates failures.
2. Certain choices of angle result in failures. This is strikingly illustrated for 1 processor, and is due to, among other things, the condition of this polynomial. Picking the first successful execution would work for any number of processors greater than 2 in this experiment.

4.9.2.2. Parallel Execution

The Jenkins-Traub algorithm displayed significant variance on this input and others. Since we had a working program available, we decided to measure the performance of our scheme on a multiprocessor system to see how much overhead arises in the complete, end-to-end algorithm. Copies of the Jenkins-Traub algorithm were run using the parallel version of *cvaryangle.c* given as Appendix XVI. *Cmach.c* was run on several multiprocessor systems. For an otherwise idle two processor Ardent Titan, the first six lines of *cjt* produced the results:

procs	max	min	avg	fails	par
1	4.01	4.01	4.01	0	4.37
2	4.49	4.07	4.28	0	4.25
3	4.45	2.03	3.50	0	4.74
4	4.48	1.37	3.31	0	5.19
5	4.27	2.36	3.35	2	8.61
6	4.50	2.02	3.65	0	7.03

The column labeled "par" is the parallel execution time measured with *cmach*. All times

below the double lines were with processor contention, since the number of processes was greater than the number of available processors.

Thus, the execution time overhead of creating two processes and running them concurrently can be computed as 4.25-4.07 sec., or about .18 sec. But the average time was 4.28 sec, so even with the additional overhead, the parallel execution finished first.

While encouraging, the numbers suggested that more processors might yield more impressive results. On an experimental [Garcia1989a] multiprocessor, *cmach* was run on the degree 16 polynomial $z^{16}+$ $(19.800000-3.200000i)\cdot z^{15}+$ $(176.930000-58.940000i)\cdot z^{14}+$ $(940.246000-494.724000i)\cdot z^{13}+$ $(3284.874700-2495.183000i)\cdot z^{12}+$ $(7821.151500-8375.778560i)\cdot z^{11}+$ $(12521.112583-19491.045162i)\cdot z^{10}+$ $(12045.424807-31405.685381i)\cdot z^{9}+$ $(2785.185034-32740.983162i)\cdot z^{8}+$ $(-10378.708048-15172.462694i)\cdot z^{7}+$ $(-16870.986162+13554.317038i)\cdot z^{6}+$ $(-12430.788642+33571.308326i)\cdot z^{5}+$ $(-3673.995257+33389.404113i)\cdot z^{4}+$ $(1360.752190+20180.264386i)\cdot z^{3}+$ $(1689.471245+7674.191752i)\cdot z^{2}+$ $(622.122192+1700.277923i)\cdot z^{1}+$ $(86.407858+168.238421i)$. An eight-processor configuration running MACH produced the results illustrated in the following table:

procs	max	min	avg	fails	par
1	12.1	12.1	12.1	0	12.8
2	13.1	6.2	9.6	0	6.3
3	22.9	13.1	17.1	1	12.3
4	13.4	6.23	10.7	0	6.4
5	19.9	5.8	12.2	0	6.3
6	23.0	6.2	13.8	1	11.4
7	16.9	5.9	10.1	0	7.4

When plotted yields:

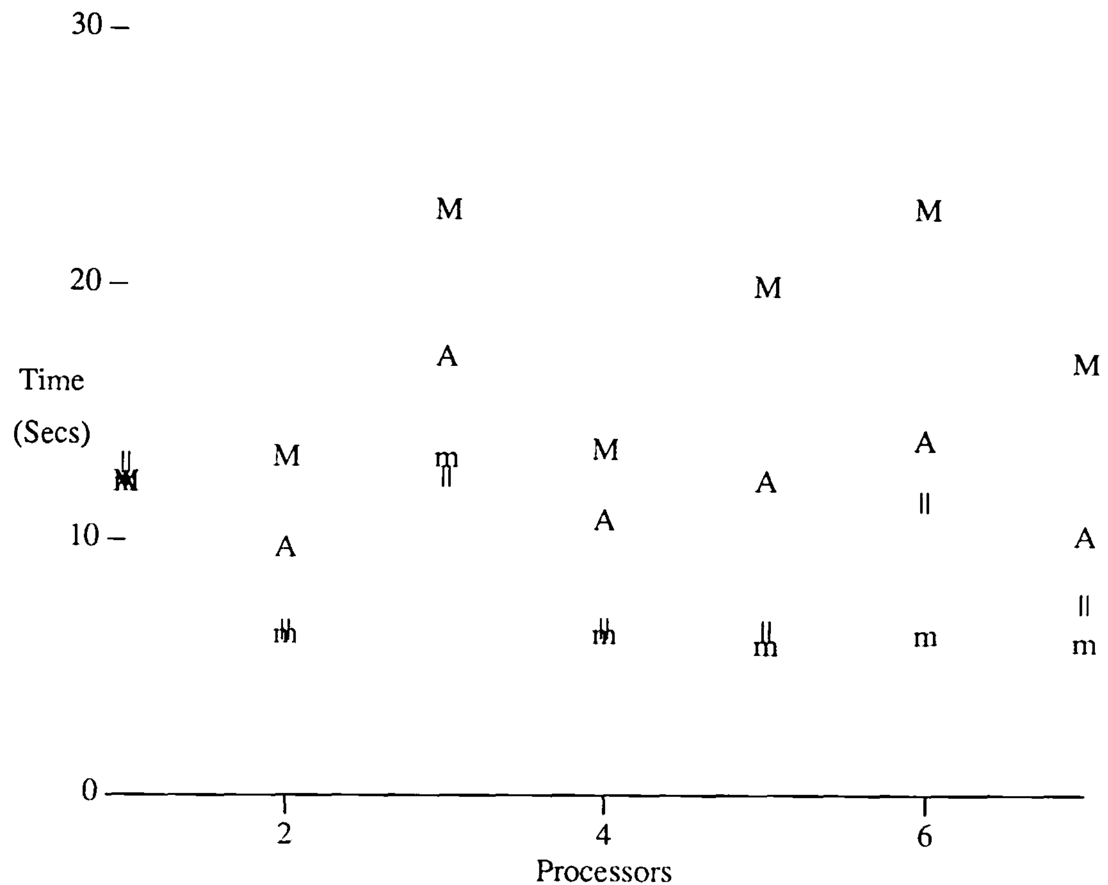


Figure 51: Performance of polynomial root-finder

The legend is quite simple: **M** designates the maximum time of an ensemble of angles, **m** designates the minimum time, **A** designates the mean time (excluding failures), and || is the execution time, including all overheads, required for parallel execution. The overhead can be calculated from the gap between **m** and **A** values in the plot. The speedup can be seen by comparing **A** to ||. This legend is used for the plots of Appendix III.

For these tests, a single process was run before the testing started, in order to pre-page the code into the multiprocessor from the host; this single process took 17.2 seconds. This was done because the ACE is attached to an IBM PC/RT host, from which code is loaded into the ACE address space before execution. This initial load would introduce a spurious execution time increase, and must be removed from the results. A variety of polynomials were tested; this was neither the best nor worst performance observed. A number of other test sets and performance tables are given in Appendix III. Figure 52 summarizes the speedups seen on these examples, which were chosen based on

the advice given in several papers on testing rootfinders. The summary statistics clearly reflect the limited dispersion available, as illustrated in the graph.

We first calculate the speedup for a given number of processors on each polynomial by computing the ratio of the average time to the observed time for parallel execution. The mean of these speedups is then computed, giving the value in the table.

procs	speedup
1	0.99
2	1.16
3	1.23
4	1.28
5	1.26
6	1.11
7	1.10

So, for example, the method produced, on average, a 28 percent speedup when 4 processors were utilized. This is a real speedup, since all processing overhead is accounted for in the observed execution times.

The research contribution of this method of concurrently executing the alternative versions of the Jenkins-Traub algorithm is significant. Certain component operations of the algorithm can be vectorized, so that processors such as a Cray [Russell1978a] or CDC 6600 [Thornton1970a] CPU could extract some parallelism from the execution. However, this parallelism is available to speed up all the alternatives, since they differ only by an initial angle and not in their executable code, which is shared. Other than exploitation of vectorizable code, or perhaps fine-grained dataflow properties, the algorithm is inherently sequential, due to the use of polynomial deflation [Ralston1978a] when a root is found. Yet, we have shown a method by which significant speedups can be achieved through use of multiple processors, and we have demonstrated those speedups in practice.

4.9.3. Other Applications of the Technique

We discuss other applications of the technique. These are drawn from varying areas of computer science.

4.9.3.1. OR-parallelism in *Prolog*

Logic programming is a new method of harnessing computers to solve problems. The major idea is that the "logic program" *declares* what must be true, and the logic programming system finds a way of making this true. This results in a solution to the problem. Logic programming is discussed by Kowalski [Kowalski1979a]. Logic programming's attraction to researchers is primarily because of the freedom from constraints on "how" the "what" of the program is to be achieved. Thus, schemes for short-cuts and parallel execution abound. Much of this research activity has had as its focus the *Prolog* language, to which we will restrict our attention.

The *Prolog* [Clocksin1984a] programming language is based on predicate logic [Kowalski1979a], using "Horn clauses" [Rich1983a] to describe data and interrelationships. Many normal operations are subsumed by the unification algorithm by which *Prolog* attempts to satisfy predicates: variables are bound during the unification process to values which caused the predicates to become true. Thus `equal(X,elrod)` will cause the variable `X` to take on the value `elrod`, as this binding is the only one which allows the predicate `equal()` to be satisfied.

Progress is achieved with a goal-oriented predicate-satisfaction algorithm; a database of predicate values and rules is used to construct a set of dependency relations; top-level goals are decomposed into sub-goals using the relations between the rules, objects, and predicates. For example, testing equality of lists implies that their elements are equal; testing element-wise equality may then give a list of sub-goals. This gives rise to a possibility for parallel execution, however the granularity of such parallelism seems inappropriate. More appropriate is rule-level parallelism, which is centered on two types, AND-parallelism and OR-parallelism. The idea with AND-parallelism is that if we have a situation where goals A and B must be simultaneously satisfied, we can pursue the satisfaction of A and B in parallel, and deal with the simultaneity later. This, however, has proven possible but difficult, due to the requirement of consistent variable bindings in A and B. The situation is similar for OR-parallelism; this is more interesting to us,

since it maps closely to our problem of attempting alternatives in parallel.

4.9.3.1.1. Tutorial Example

We'll start with a database of "facts," which are essentially ground literals of first-order predicate logic. The example we'll construct will be useful in understanding the *Prolog* execution engine, so that we can examine what opportunities exist for parallel execution. The knowledge base consists of the following facts:

```
father( charles, jonathan ).
father( frederick, charles ).
father( raymond, jacqueline ).
father( john, gertrude ).
father( charles, steven ).
mother( jacqueline, jonathan ).
mother( gertrude, jacqueline ).
mother( julia, charles ).
mother( mary, gertrude ).
father( frederick, elizabeth ).
father( frederick, frederickjr ).
father( frederick, patricia ).
```

First, let's consider some queries in the sequential execution of a *Prolog* interpreter. Those desiring a more detailed tutorial can consult Clocksin and Mellish's [Clocksin1984a] book. The clause

```
grandfather( X, Z ) :-
   father( X, Y ), father( Y, Z ).
```

defines a predicate which defines grandfatherhood in terms of two related fatherhoods. The interpretation of the comma is that it combines the clauses separated by it in such a way that all of them must be true. The ":-" can read so that the right-hand side *implies* the left hand side. Thus, both `father( X, Y )` and `father( Y, Z )` must be

true for `grandfather( X, Z )` to be true, and `Y` must take on the same value in both clauses. Consider the *Prolog* query:

```
?grandfather( G, jonathan ).
G=[frederick]
```

What has happened here is that the *Prolog* interpreter has taken the query and decomposed it into the two sub-queries. It first tries to match `father( X, Y )`, and the first match (assuming the literals are organized as shown) in the database will be the literal `father( charles, jonathan )`. Thus, `X` is instantiated as `charles` and `Y` is instantiated as `jonathan`. The second clause must then match `father( jonathan, jonathan )`, which it does not. Since the match failed, the algorithm backtracks to the last *choice point*, and the search for a value of `father( X, Y )` continues, and `X` and `Y` are instantiated as `frederick` and `charles`, respectively. Since `Y` has now been "bound" to `charles`, the database is examined for a match for `father( charles, jonathan )` which it finds as the first entry. The interpreter then sets `G` to `frederick` and prints the value of `G`. The interpreter also waits for an indication that the search is to be continued, in case there are more solutions to the query[16].

Now, we have ignored the possibility of a maternal grandfather, which can be specified with the additional clause

```
grandfather( X, Z ) :-
    father( X, Y ), mother( Y, Z ).
```

The existence of this clause changes the interpreter's control flow in the following way, as a result of there being two alternative methods of demonstrating grandfatherhood. Consider the query:

```
?grandfather( raymond, jonathan ).
T
```

There is no pair of values which satisfies the first clause, so it would fail. However, there remains a second clause which can be tried, and `father( raymond, Y )`,

[16] This presentation is more concerned with the variables and control mechanisms than with the full unification theorem proving scheme; thus our portrayal of the process as "pattern-matching" is sufficient.

`mother( Y, jonathan )` is satisfied by `Y=jacqueline`, and the query is satisfied.

The first query offers an opportunity for parallelism in the database search. Elements satisfying the first sub-query can be matched against elements satisfying the second; this merging process produces a result which can satisfy the original query. If the conjunction of clauses does not share variables, then the merging is easier. Management of the general case of these queries is difficult, since the sub-queries may have caused shared variables to exhibit side-effects, i.e., "bindings." However, in having two methods for "proving" `grandfather( raymond, jonathan )` we have introduced a new opportunity for parallel execution. The *query* is true if *either* top-level clause results in an answer. Thus, we can attempt to process both of these queries in parallel, since a successful sub-query in either case can cause the top-level query to be satisfied.

One major problem, as pointed out by Warren in his survey [Warren1987a] (ignoring side effects) is the problem of "multiple binding environments." What this means is that as the clauses execute in parallel, they bind values to variables, e.g., `jacqueline` to `Y`. Now, different alternatives may bind different values to the variables at different times, and hence interfere with each other, in a way that would not be possible in the single threaded stack-oriented interpretation style we described. How this problem can be dealt with is a question which has inspired many researchers, and engendered many solutions. A good, but somewhat dated, survey is provided by Wise [Wise1986a] who surveys work in parallel logic programming for comparison with his EPILOG system. Some of these are described in the next section.

4.9.3.1.2. Existing Solutions

The basic problem is to prevent the clauses from interfering with each other, through side-effects, during their execution. One solution which has been examined is called *committed-choice non-determinism*. Clauses are partitioned into *guards* and *bodies*, so that they take the form:

```
head :- guard, body.
```

The guards of all clauses selected by the matching algorithm are evaluated in parallel. From among the clauses whose guards are true, a body is selected for execution. The

choice of this clause is the *committed-choice*. In order that they can be executed in this fashion, the guards are often [Clark1987a] constrained to be *read-only*, i.e., side-effect free. For example, static analysis by the PARLOG [Clark1987b] compiler insures that the clauses are read-only, before execution. The GHC [Ueda1987a] scheduling mechanism blocks processes which update shared state.

Another approach is maintain the language features of *Prolog*, and introduce the parallelism in a transparent fashion. This is the approach of the systems Warren describes in his survey. They use a combination of interpreter-controlled variable management and interpreter-controlled process scheduling to prevent updates which would result in an inconsistent state. Processes are typically modeled as independent threads of control working in a common address space. Thus, we could look at the *forked()* sub-processes as inheriting the state of their parent process; classical resolution[17] would copy all the state at each sub-node in a computation tree. A "node" is this tree is a list of goals; for example the root node's goal is satisfaction of the initial query. The computation consists of a series of decompositions of the initial query into clauses which are satisfied, left to right in a traditional *Prolog* system. This results in a depth-first search of the tree; search termination is achieved when a goal is "resolved," or matched with some entry in the knowledge base. As each sub-node is executed, it consists of a copy of its parent node, with the left-most clause instantiated to whatever knowledge base entry matched it. Aside from the execution cost, the main additional overhead here is the copying of the parent's goals. This burden of copying would occur at each branch point, where a *fork* would occur, in a parallel execution. Aside from this burden, Warren points out that the approach does offer the attractive properties of (1) the processors can work independently on physically separate data, and (2) in all other respects, the implementation would be the same as a sequential implementation, and therefore would be able to take advantage of any optimizations, etc., that might apply.

A simple modification, and an optimization, is the "Naive Model," where rather than copying state, binding lists (<name,value> pairs) for variables are associated with each node in the tree. The sub-node instantiates the elements of the binding list; no physical copy need be made of an unchanged variable. To look up a variable's value, a

[17] Warren's survey provides a much more in-depth discussion of what he calls the "Abstract Model of classical resolution;" see also the discussion of resolution in Nilsson [Nilsson1980a].

search through the binding list must be made, as bindings form a LIFO queue. The major drawback is that the size (and hence the search time) of the binding list is not bounded. The ''Naive Model'' is close to the way sequential *Prolog* maintains its trail stack, used for backtracking from a failed goal.

To address this problem of unbounded lookup time for a binding list, the ''binding array'' was proposed. The idea is that each processor has a binding array in local store; there is one node per processor. The array is instantiated when a goal is selected for resolution by a processor; it is a constant cost lookup, so that the ''binding list's major problem is eliminated. A clever implementation uses the binding array data structure to ''shadow'' the contents of the binding list, as goals are executed. When a processor begins to execute a new goal, the binding array must be set up, so that task switching now gains more overhead; this can be considerable, depending on how many bindings must be changed.

The ''Argonne Model'' uses hash table lookup to cut down the cost of searching the binding array by a constant factor. There is in addition the idea of ''favored'' bindings, which are local to the processor, and ''unfavored,'' which are non-local in the sense that they are bindings that are ''favored'' by some other processor. Non-shared variables are denoted ''private.'' Only ''unfavored'' bindings must be looked up in the hash table, although hash tables are created for each arc of the OR-parallel tree, and are kept in a linked ''chain.'' Figure 4 of Disz, *et al*'s paper [Disz1987a] is particularly illustrative. ''Private'' and ''favored'' variables have constant times for binding (i.e. assigning) values to variables. ''Unfavored'' bindings are more costly, as they involve table accesses and updates. If most shared variables are ''favored,'' this scheme is effective. However, as Warren points out in his survey [Warren1987a], the fraction of variable references which are unfavored is high, i.e., 0.2->0.5. Disz, *et al.* [Disz1987a] were similarly disappointed by the performance of the scheme on benchmarks which they reported. A ''context-switch'' (where the processor suspends execution of the currently executing process, and resumes the execution of some process) is expected to be cheap compared to the context-switch cost for the ''binding arrays'' scheme, as it involves only establishing pointers to the proper hash chain, not copying. For the ''unfavored'' bindings, the lookup cost, while reduced by a large constant due to the hashing technique, can still be (potentially) unbounded, as it remains proportional to the number of unfavored bindings.

The "Manchester-Argonne" model improves upon the Argonne model by applying the observation that the Argonne model's hash chains are not necessary unless sharing occurs. Thus, in the optimized model, allocation is postponed until a processor wants to share an "arc;" at this point the number of relevant bindings is known and assuming a reasonable "randomizer" the hash table technique can yield a constant-bounded time lookup, and Warren argues that it will, with scheduling and merging techniques which he describes.

Finally, the "Argonne-SRI" model uses the "binding array" technique of the "SRI" model combined with the favored/unfavored distinction between shared variables from the "Argonne" model. The private "binding array" is used for unfavored bindings only; "favored" bindings are marked with some flag indicating their status; this flag is associated with the value. (Note that the space for this flag must be available in *all* value cells. If the "space" devoted to this is greater than the fraction of shared variables, the straight "binding-array" may be more space-efficient.) Thus, if many bindings are "unfavored," this technique will cause an extra access (of the binding array) for each of them, but the time will be constant. Warren's opinion of the merit of the various schemes is summarized as:

$$Abstract<Naive<\begin{matrix} Argonne<Manchester\text{-}Argonne \\ SRI<Argonne\text{-}SRI \end{matrix}$$

Now, all these schemes make the following assumptions:

1. The *Prolog* implementation (compiler/interpreter) must manage its own memory.
2. If multi-processing, memory is shared.
3. The *Prolog* implementation is responsible for task management.
4. Variable-binding is considered to be the major problem. The implication of this is that other concerns, e.g., side-effects and IPC, are ignored or given short shrift.
5. Adherence to sequential *Prolog* syntax and semantics must be maintained. This is a more modern version of the "dusty-deck" arguments associated with any changes in an existing language.

Unfortunately, these assumptions are not always met, or in some cases, reasonable. We will examine the assumptions point-by-point.

1. Unless a special-purpose *Prolog* machine is built, and in addition is economically viable, the cost/utility/performance of virtual memory will dictate its inclusion in general-purpose computers. Thus, in reality, an abstract "memory reference" must go through *two* memory management schemes: *Prolog*'s, and the hardware's virtual addressing support (this ignores other "transparent" architectural features such as caches).

   An interesting question which has arisen in the study of block sizes (see Deitel [Deitel1984a] for a discussion) for page-oriented virtual memory systems is the penalty paid for fetching data in blocks which may or may not be well-correlated with the problem characteristics. For example, if accesses are done on memory-word sized units which are widely separated, each word access may require several thousand instructions to handle the associated page fault; the same problem arises when choosing a cache line size. While this difficulty can be reduced, if not eliminated, by problem-specific approaches such as overlays, these are clumsy to manage and are not transparent to the programmer. In any case, most programs exhibit *locality of reference;* LISP and *Prolog* programs are no exception.

2. While shared-memory architectures are common and provide a nice abstraction for the programmer, distributed computing is different with respect to the latency and bandwidth of communication versus the latency and bandwidth of local memory referencing [Smith1986a], therefore the economics of communication become much more interesting, even if the communication is buried beneath a shared memory abstraction [Abramson1985a, Delp1988a, Cheriton1986a, Li1986a, Li1986b, Stout1983a, Poplawski1987a]. For example, *locality of reference* becomes much more important because its *impact* on performance is so much greater. Some attention has been paid to the issue of *Prolog* memory referencing behavior in the literature. Tick's [Tick1988a] thesis provides a wealth of detail, but assumes a specific model for the abstract machine. Nguyen [Nguyen1988a] uses a variety of benchmarks for a *Prolog* running on an existing system, some of which are discussed later in this section. Ross [Ross1986a] discusses detailed measurements of virtual memory access behavior for a *Prolog* system, and makes several suggestions for improved performance.

   While attractive in terms of reducing unnecessary references, indirection and pointer-based <name,value> retrieval look less attractive when the *number* of remote

references is calculated. Crammond [Crammond1985a] looks at three methods of creating an execution environment for parallel execution of disjunctions. He provides some analysis of mechanisms designed for efficient reference of shared data, in particular the update of shared data. He studies three algorithms, which he calls "Directory," "Hash Window," and "Variable Importation." "Directory" and "Hash Window" do bindings (i.e., variable instantiations) on demand, via pointers, while "Variable Importation" imports everything it may need, at higher initial overhead, but taking more advantage of locality once it performed the importation. Crammond points out in his study that copy-based schemes ("variable importation") for multiple binding environments, such as Lindstrom's [Lindstrom1984a, Lindstrom1984b, Tinker1987a] begin to look more attractive under such circumstances, since their performance is competitive with the other schemes in any case.

3. This is not a bad assumption in intent, as the *Prolog* implementation's knowledge of semantics and scheduling heuristics (which are not applicable to more general-purpose operating system process schedulers) may make it a much more effective scheduling tool. However, if interaction with the scheduler of a general-purpose operating system is necessary, as we argued it would be without *Prolog* machines, then heuristics will be applied based on CPU utilization rather than execution-time performance, (they may be different, as we saw earlier in our measurements of overhead) and due to load-balancing schemes using process migration [Ferguson1988a], not easily predictable. Communication between the scheduling component of the general-purpose operating system and the *Prolog* implementation thus seems like an effective strategy in these circumstances.

4. One major problem which the committed-choice non-determinism logic programming languages [Shapiro1986a] address is that of side-effects. How is an "all-solutions" requirement dealt with if each solution writes a payroll check? Butler, *et al.* [Butler1988a] point out how these issues damped their initial enthusiasm [Ciepelewski1985a] for the "Dusty Deck" solutions, and led them to include a "commit" operator in their language [Hausman1987a, Lusk1988a]. While it could be argued that the programmer must be aware of such possibilities and adjust for them, this seems wrong, as experience has shown that good design of support mechanisms enhances programmer productivity, and myriad subtleties and

restrictions reduce it. ``Committed-choice'' semantics address the issue simply: they guarantee that only one result will be produced by the construct, and do not specify the selection criterion in order to allow themselves full use of the degree of freedom provided by nondeterminism.

5. Cost-benefit analysis usually dictates whether changes can reasonably be made, if they are technically justifiable. When a large amount of software assuming a certain semantics is written, this large installed base and its assumptions represents a huge cost which must be weighed against improvements derived from the change. For example, the performance benefits of pipelining in the IBM 360 Model 91 were considered significant enough so that the architectural specification of the 360 for that particular machine was changed to include an *imprecise interrupt*. *Prolog* has the advantage in that it is newer, and the installed base is much smaller, and that a small change can yield such benefits, especially when considered in the context of Point #4. In addition, the ``guarded command'' structure has strong intuitive appeal, as it has appeared independently in several settings [Dijkstra1976a, Randell1975a, Ledgard1981a].

4.9.3.1.3. Discussion

What our method does is copy using virtual memory support provided by the operating system, and since we choose only one alternative, no merging is necessary. Since there are no extra (beyond whatever is required for sequential execution) pointer chains to traverse on variable references, memory access is fast. Use of the method requires changing the *Prolog* interpreter to detect and exploit OR-parallelism. How aggressively available parallelism is exploited is a function of the overhead associated with maintaining a process. However, once this is known (we show how to measure the overhead in Sections 4.2-7 of this thesis), the proper granularity can be used as a factor in the decomposition process.

We see the following advantages to our approach:

1. Virtual memory support in some form is going to be available on general purpose computers, both because it is useful, and because it makes a great deal of economic sense. While *Prolog* machines are desirable, as were the LISP machines, it's not clear that the performance advantages which accrue from special-purpose

architectures can overcome the steady increase in performance seen by general purpose computing engines.

2. Our arguments for performance were completely general, and our example applications have been drawn from a large spectrum of computing tasks. Thus, the mechanism itself is useful outside of *Prolog*, and thus should be made available to other applications.

3. The workstations our measurements were made on are representative of the processing components which comprise many of the commercially- produced multiprocessor engines. For examples, see the Sequent [Beck1985a] or Encore [Encore1985a] multiprocessor machines.

4. The problem with copying is that it's costly. But, (1) as we've shown in this thesis, the cost can be parameterized (2) the cost can be reduced through optimizations such as copy-on-write (3) once the copy is made, it takes full advantage of any available per-processor cache [Smith1982a]. Following a pointer chain or indirection through a hash table will then cost more after the first reference. Analysis of the page-fault cost amortized over subsequent references versus the cost of following chains (much of which may be cached as well) on each reference should be done, although the analysis of Crammond, as discussed above, points to the fact that one implementation of a copying scheme (not page-level) did not perform badly compared to indirection-based schemes.

5. When considering a distributed system, as opposed to the bus-connected shared memory multiprocessor which most studies have assumed, the cost of remote references is increased to the point where copying looks increasing attractive, as the copy which is made serves as a cache.

6. The support of *Prolog* OR-parallelism entails certain overheads even in interpreter-based memory management schemes. While the more advanced schemes described above tend to take a more demand-based approach to copying, there is overhead associated with adding these features which is incurred by all *Prolog* programs running using the augmented interpreter, e.g., extra tables, extra pointers, or restricted syntax. Our method takes advantage of services already available under general-purpose operating systems, and a *Prolog* program which does not use the features runs no more slowly than the same *Prolog* program running on a system without the features.

It must be noted that the interpreter-controlled approaches which use their own tables to handle multiple binding environments will be portable, while our virtual memory based scheme will require interaction with an operating system, which may have an adverse impact on portability. However, as we've argued, and as others have observed, solving the multiple binding environments problem is not equivalent to managing all side-effects.

In summary, we've offered an interesting and sometimes advantageous scheme for executing *Prolog* programs in parallel. In addition, since we have quantified where the advantage will occur, measurements of program execution time combined with our overhead figures from previous sections of this chapter will give us a good estimate for the technique's effectiveness.

4.9.3.1.4. Measurements of Published *Prolog* Programs

To test the efficacy of our technique, we measured some programs from a standard *Prolog* reference, Clocksin & Mellish [Clocksin1984a]. The programs are different algorithms for sorting, applied to various data sets. The sources for the sorting programs are given as Appendix IV. The programs were applied to three data sets, those of Appendices V, VI, and VIII. The data of Appendix VII was used to indicate the growth rate of the execution time with the data set size for a naive ''generate-and-test'' sorting strategy. The results of sorting the large data set of Appendix V are shown in the following table.

Sort Name	CPU Sec.
qsort	24.4
quisortx†	5.283
quisort	41.1
insort	6097.0
busort‡	65.9
sort	∞

These results show an incredible amount of variation, although the strategy would prefer

† Failed, no explanation.
‡ Failed, stack overflow

`qsort` on this size of list. The naive sort took 2811.8 seconds of CPU time for 9 elements (`X38`) which is indicative of why it has so much trouble. If the "random selection" picked this algorithm for a large list, execution time would be infinite for all practical purposes. Smaller lists exhibit more variation. For example the small ordered list benchmarks are shown in the next table.

Sort Name	CPU Sec.
qsort	0.267
quisortx	0.267
quisort	0.233
insort	0.100
busort	0.083
sort	0.133

Ordered lists were chosen because the implementations of quicksort are known to have trouble with them, and for this task it is obvious. The win on a small "random" list is not so dramatic, although there is some variance:

Sort Name	CPU Sec.
qsort	0.117
quisortx	0.133
quisort	0.117
insort	0.117
busort	0.633
sort*	∞

These simple tasks used a significant amount of CPU time, and the cases that exhibited

* After 21,500 minutes (= 358 hours, or more than 14 days) of CPU time (HP9000/350) this process had not printed a result. I gave up.

usable variance used little memory. For the larger lists, more memory was used, implying more copying. This is due to the deeper recursion.

Nguyen and Despain [Nguyen1988a] have compiled a series of tables containing memory access statistics for *Prolog* programs applied to standard benchmarks. The statistics indicate that much of the traffic is to stack data segments (Stack+Trail) (min=14.6% for cmatch_strA , max=77.9% for cdeep_bakA, mean=46.1%); the rest is to the heap (mean=23.4%) and H2. H2 is used for control of parallel execution. Other statistics breaking the accesses down into reads and writes show that on average, the benchmarks **write** to the heap about 7.7 percent of the time. What these results indicate is that for almost all the benchmarks, there was more write traffic to the stack(s) than to the heap. This is important to our method, since stack writes will occur after the `alt_spawn()`, and hence will be local to whatever processor the process is executing on, rather than causing the extra traffic caused by the copy which must be made when the heap is written to. Writes to the heap are the major source of copying in the address space model used by our method. Thus, the copy-on-write technique can save by a factor of $\frac{100}{7.7}$ or about 13 in the amount of state which must be copied. Thus, we expect the combination of ''copy-on-write'' and ''fastest-first'' OR-parallel execution to produce a significant speedup for many programs.

4.9.3.2. Polyalgorithms

Polyalgorithms [Rice1968a, Symes1971a] have been suggested as a method for encapsulating a numerical analyst's knowledge into a system for solving numerical problems. The basic idea is that several methods are combined [Rice1969a] along with information about the circumstances under which a method is likely to be successful. As different methods are tried and fail, information about the problem is built up until either there are no successful solutions, or a solution method succeeds (for example, discovering multiple zeros in a failing root-finder may be useful to the next solution method).

Our scheme could be used by creating artificial ''alternatives'' with the available solution methods. Each ''alternative'' trys a different solution method ''first,'' to create alternative versions of the polyalgorithm. ''Fastest first'' scheduling could improve the response time properties of a system such as NAPSS [Rice1973a], especially since the performance of the system was perceived to be a problem [Rice1988a].

We should note that Traub [Traub1964a] mentioned such an idea (exploring multiple solution methods) as a direction for future work in his book on iterative methods.

4.9.3.3. Simulation

Simulation tasks are attractive applications for speculative methods, as some success has already been demonstrated by Jefferson [Jefferson1985a, Jefferson1987a], in his "Virtual Time" scheme, which uses rollback-recovery to mask unsuccessful speculations. Note that we delete unsuccessful speculation by deleting the processes which contain its results. Consequently, we never roll back.

5. Related Work

Exploring alternatives in parallel is far from a new idea; hardware engineers looked to it as a way of maintaining pipeline[18] utilization in some high-speed computers, most notably the IBM 360 Model 91 [Anderson1967a]. Their approach was to prefetch components of both possible branch paths until either the results of the conditional execution are available (in which case the correct stream can be chosen and the other discarded) or an irreversible side effect (such as instruction execution) would occur. One possible approach is to prefetch components of both possible branch paths until either the results of the conditional execution are available (in which case the correct stream can be chosen and the other discarded) or an irreversible side effect (such as instruction execution) would occur. This approach is analyzed by Riseman and Foster [Riseman1972a], who conclude that speedups are possible if the correct branch is taken; their results are from a set of 7 programs run on the CDC 3600 computer. In the architecture setting, there is the problem that hardware is needed for each execution path and all paths must be explored, leading to a combinatorial explosion which affects even small values of the number of conditionals. Our management of side effects lets us go further.

Bernstein [Bernstein1980a] points out two problems with Hoare's CSP proposal. The first is that a programmer may want to have some method for choosing between alternatives with open guards; he solves this with priorities. The second is that CSP guards cannot contain output expressions; Bernstein solves this by use of the fact that the I/O primitives name a specific process; thus, guards specifying that process can be delayed until the specified process is ready. This is essentially a scheduling trick.

Version control systems such as SCCS [Rochkind1975a] and RCS [Tichy1985a] use the idea of deltas to store multiple versions of data. More related to our *predicates* is the idea used in the PEDIT [Kruskal1984a] parametric line editor. Associated with each line

---

[18] A *pipeline* in a computer architecture is a logic implementation based on the principle of an assembly line. That is, there are a series of specialized functional units which solve a portion of the problem; these can be made rapid, and ideally all the functional units can be kept busy at the same time. Analogously, the ideal case is realized when everything is predetermined. As Henry Ford said,

> "You can have any color you want, as long as it's black."

Difficulty is caused by conditional statements which can take one of two paths, thus inhibiting pre-fetching of instruction components. There are several approaches to resolving this difficulty; Lilja's [Lilja1988a] survey discusses the strategies.

of text is a set of *parameters*, hence the name *parametric* editor. These parameters are state variables, e.g. `SYSTEM=UNIX`, `VERSION=SysV`, et cetera. The line is selected for display if the mask set in the view of the file matches the settings of the state variables; thus, the viewer of a source program in a particular environment might see the source without the obscuring effect of various conditional compilation directives. Each setting of the state variables gives a distinct *version*, but in practice most of the text is shared between the versions.

Our method uses predicates to detect conflicts, but delays their resolution as long as is possible. Thus, it is *optimistic* in the sense that each timeline assumes that it will succeed. At each point where this success may come into question, it generates a predicate. These predicated processes are similar to the *possibilities* and *dependencies* discussed by Reed [Reed1978a] in his thesis; however, his NAMOS system was further from realization than the methods described here.

The notion of multiple alternatives is orthogonal to the transaction concept; if we view an alternative ``block'' as effecting a transaction on the system state, the specification is a description of how to accomplish the transaction reliably. It could also be viewed as a set of ``competing'' transactions, at most one of which will take effect.

One significant feature of our use of *predicates* is that there is as little *waiting* as possible in the system: each process which could only execute under a set of assumptions simply makes that set of assumptions, until some conflict with the correctness policies results. In other settings, such methods are called *optimistic* [Kung1981a, Strom1987a] because they assume that delay-causing or failure-causing conditions happen infrequently. Thus, normal operation is made cheap, at the expense of more expensive handling when the assumption is wrong. In our setting, the operant *optimistic* assumption is that the executing alternative is the one which will complete successfully. Thus, the predicates indicate that a process assumes that it will complete successfully; rather than *waiting*, it *continues under that assumption*. Strom and Yemini's [Strom1985a] *dependency vectors* behave much like our predicates.

Distribution of computation across several nodes offers attractive possibilities for both reliability and performance. Cooper [Cooper1985a] discusses the use of replicated distributed programs to take advantage of this potential. Cooper's CIRCUS [Cooper1984a] system transparently replicates computations across several nodes in

order to increase reliability. Goldberg [Goldberg1987a] has also discussed process replication, with a focus more on performance than fault tolerance. Replication is different than the problem we have examined, mainly because we cannot count on all the concurrent alternatives exhibiting the same behavior, e.g., reading and writing. For example, when managing I/O for replicated computations, only one read operation can be performed, and its results buffered for subsequent readers of the same data. Thus, idempotency of some *source* state can be forced through buffering.

Transparent replication can easily be combined with the use of parallel execution of several alternatives for increases in performance, reliability, or both.

6. Conclusions

This thesis has carefully examined problems for which there are multiple alternative solutions. When all such solutions are equally acceptable, the strategy of executing all alternative solutions concurrently has some benefits. In particular, this thesis has examined selecting the first successful computation as an approach to improve response time. We described scheduling strategies and memory management policies to ensure consistency and correctness. To restrain the growth in state required for concurrent execution of "Multiple Worlds" we suggested the use of "copy-on-write" storage managed in units of pages. External interactions are controlled by (1) an extension of the page management scheme to include slower storage, i.e., files; and (2) an extension of the page management scheme to interprocess communication, where interacting processes are also "copy-on-write."

The results of this thesis are dependent on achieving an improvement in response time. We first identified the overhead of concurrent execution using the algorithms of Chapter 3. Using this model of overhead, we provide a measure of the performance improvement (*PI*) which can be used to compare execution strategies. We use the measure of performance improvement to identify opportunities for response time improvement. The best situation (discussed formally in Chapter 2) for the approach presented here is one where:

- Alternatives require a significant amount of computation time, as encapsulated in $\tau(C_{mean},\vec{x})$.
- Each alternative changes a small amount of the state of the calling process, thus reducing the penalty $\tau(overhead)$.
- There is enough difference between the execution times of the alternatives that choosing the fastest and killing the others is worth the overhead of spawning the copies and deleting the slower siblings.

We made the observation that the speedup is dependent on both the execution time devoted to overhead and the dispersion exhibited by the execution times of the alternatives. The implication is that if overhead can be understood and controlled, there is an opportunity for speedup roughly proportional to the dispersion; thus superlinear speedups, where the execution time is less than $O(\frac{1}{N})$ for N processors, are *possible* under this

scenario.

Since the potential for speedup is sensitive to overheads, we examined these in Chapter 4. Copying is the major overhead in the creation and maintenance of concurrent executions. There had been no previous work examining the efficacy of "copy-on-write;" our results indicate that the technique is extremely effective in practice. The thesis provides several useful measurement tools to gather execution time data. These tools are used to adapt the measurements we reported here to new computers; thus, with a few measurements the domain for performance improvement using this method can be delimited. We have implemented a system which performs a "remote fork," which is similar to *process migration* with the exception that two processes exist when the operation is successful, rather than one migrated process. The measurements confirmed that the major cost in overhead was copying, and further, offered empirical proof that child processes could be "spawned" in a distributed execution environment. Response time is affected by the response times of several system components; we examined two subcomponents, the disk and the network, in Chapter 4. The final overhead was "sibling elimination," which we modeled on a multiprocessing timesharing system. This should represent the worst case execution, and the results were encouraging, as they show the elimination to be remarkably cheap, and insensitive to process behavior. Thus, they are predictable and can be used in deciding whether to apply the concurrent execution scheme.

After identifying the opportunities with analytic work and measurements, some applications were. Distributed Execution of Recovery Blocks and parallel execution of logic programming languages are well-matched to the semantics of our execution scheme, and both can benefit from the higher performance. In each case, there are some restrictions imposed by the setting. Parallel implementation of logic programming languages provides an appropriate environment, because the computation is data-driven, and thus the execution time and control flow can vary greatly with the input. The way in which unification operates (as a "sophisticated pattern matcher") leads to an overwhelming preponderance of read references made to page-managed memory; while a high *percentage* of references are writes, these are mainly to the stack, and thus locality should be high; stack "growth" can be handled locally, reducing copying. Many logic programs have a great degree of parallelism, so that appropriate opportunities must be identified with respect to the overheads implied by our scheme. In particular, coarse-grained parallelism is better to exploit than fine-grained parallelism. Major advantages

of the "Multiple Worlds" scheme are that (1) it deals with side-effects other than variable binding, and (2) it can run efficiently on general-purpose hardware.

Distributed execution of recovery block alternates uses the "fastest-first" behavior in an attempt to find a rapid failure-free path through the computation. The restriction imposed by recovery blocks is that they are designed for fault-tolerance. Thus, there may be further requirements beyond fast execution time; by using our RB language and suggested modifications to the concurrent execution scheme which increases the amount of state available in a system to facilitate recovery.

Numerical Analysis is rich with examples for which the choice of solution method or the parameters of a particular solution method are free variables. The Jenkins-Traub polynomial root finding scheme was executed in parallel on multiprocessors: both speedup and fault-tolerance were observed across a selection of polynomials. The distribution of execution times did not show sufficient dispersion for a linear speedup, but the results were extremely promising. In particular, the overhead of the prototype "Multiple Worlds" execution scheme was very small for this problem.

We also briefly mentioned applications to simulation. These example applications are from vastly different areas of computer science, and illustrate the general utility of the scheme. Throughput is clearly traded away for improved response time in our scheme, since the number of processors doing useful work (work that will eventually be used) at any given time will be ≤ 1. This method may be particularly useful in real-time systems, where the bias is clearly towards response time, and the sibling elimination can be carried out asynchronously with respect to result delivery.

A question which remains is the effectiveness of exploring alternatives in parallel (which is *speculative* computation using *serial* algorithms) versus other methods for utilizing available processors. Now, for a problem which is easily divided into subproblems (although this division may not be complete, and thus the sequential portion will be subject to the "Amdahl limit"), the parallel execution will most likely show an improved response time proportional to the number of processors, which means that our method will improve performance when

$$\frac{\tau(C_{avg},\vec{x})}{\tau(C_{best},\vec{x})} > \frac{\tau(C_{avg},\vec{x})}{N}$$

for N processors.

Does this happen? It's clear that it *can* happen, through choice of a particularly poor algorithm to parallelize. To focus the discussion, let's look at an example.

Problem: Given two sorted lists, L1 and L2, find an element in their intersection.

Operators: There are two operators available for the problem solution. Operator element(L,i) returns the i -th element of L, or an indication of failure if i is greater than size(L). A call to element(L,i) takes 1 unit of time. Operator search(L,x) is a boolean operator which returns success when x is an element of L, and failure otherwise. Since L1 and L2 are sorted lists, for a list of size N, search(L,x) requires time $O(\log_2(N))$: to ease calculation we can assume the calculation takes exactly $\log_{10}(N)$.

Algorithm F For each x in L1, search L2 for x.

Algorithm B For each x in L2, search L1 for x.

If there are two processors, the data parallel algorithm will apply Algorithm F (or Algorithm B; it's clear that it doesn't matter) to two half-lists. The two alternatives, Algorithm F and Algorithm B, are run in parallel. The following table compares the performance under various size assumptions for the lists:

size(L1) size(L2)	10 100	100 100	100 10
Parallel	10	100	50
F	20	200	100
B	100	200	20

Of course, the operators are rather constrained; for example it's clear that if testing for the size of the lists was easy, then the smallest list would always be used for the linear search.

There are several issues which concern implementation on existing processors, or processors similar to these. First, it might be argued that memory contention caused by several algorithms accessing the same data simultaneously will cause shared memory multiprocessors or like architectures to be slowed. We expect that this will not be true in

practice. There are three reasons:

- There is typically a cache memory per processor in such a system, to allow the existing memory technology to support more processors.
- Once the first reference is made which forces a copy of the storage, the storage will be local and accessed through the local cache.
- When the instruction stream requires memory access on a cache miss, there will be less temporal locality exhibited by the distinct instruction streams. Thus, the likelihood of undesirable interactions between the degree of parallelism and such memory properties as the cache size, the memory interleaving factor, and the cache line size, are reduced. Whatever locality is exhibited by the algorithms can be taken advantage of in the local caches, while there will not be problems if the parallel processing is slightly out of lockstep, causing repeated ''miss-fill-replace-miss'' sequences to occur.

These caching arguments can hold in a distributed setting as well. For example, *rfork()* used UNIX systems connected by a network file system. When shared memory is accessed over a network, each child process will be a ''client,'' and there will be one ''server,'' the parent or spawning process. As the children begin executing, state will be copied to their processors as necessary. If the parent has accessed the instructions or data, the state is likely to be in a buffer cache in main memory; such caches are useful in maintaining performance [Nelson1988a] in distributed systems. Main memory caches of disk objects are most often managed in LRU fashion, so that temporal locality will result in common requests being serviced by the cache. The essence of the cache argument is that if the clients access different data, there will be less contention. If they access the same data, and there is more contention, fast caches will resolve the performance problem. This effect will hold true in networks, where main memory serves to cache slower portions of the storage hierarchy.

Thus, we expect that the contention is at most only slightly worse for the MISD case than the SIMD case. The point of this is that if there is an opportunity for superlinear speedup in the execution times of the alternatives, we will not be prevented from exploiting it by some architectural difficulty. This means that applying N processors to a problem can give superlinear speedup if the execution time variance is sufficient, i.e.,

$$\tau(C_{best},\vec{x}) < \frac{\tau(C_{mean},\vec{x})}{N}.$$

and that the execution time overhead of this approach is comparable to SIMD parallel approaches. The speedup is a property of the algorithms involved and the availability of processing units. If there are fewer processors than algorithms available, the ''speedup'' must be adjusted to reflect multiprocessing or delays if scheduling is non-preemptive. If there are exactly as many processors as needed, the speedup follows our analysis. If there are more processors than algorithms, there will be unused processors which some other form of parallel execution might be able to take advantage of. However, in several applications that we examined, the number of available tasks can be controlled dynamically. OR-parallelism in *Prolog* can keep the number of tasks proportional to the number of processors by using the *Prolog* interpreter to schedule sub-computations. The Jenkins-Traub algorithm's non-deterministic component allows for an arbitrary number of alternatives to be created.

7. Directions for Future Work

Future work can proceed from this thesis in several significant directions. In the theoretical domain, the relationship between the distribution of execution times and the speedup should be analyzed. For example, the properties of an execution time distribution necessary for a given speedup could be very useful; experimental work would determine how such a distribution might be generated.

Algorithmic work and analyses should compare optimistic and pessimistic strategies, e.g., by predicting abort costs, the number of aborts, and estimate delays due to waiting on locks. The synchronization scheme should be compared to other commit/synchronization mechanisms. Some measure of cost, such as the number of messages, could be used to refine and optimize the protocols.

Page-oriented management of binding environments for *Prolog* programs should be compared to the variable-oriented schemes which are currently used to maintain multiple binding environments.

The technique does not preclude having an alternative identify itself with a state variable. Systems could be constructed which apply the concurrent execution of alternatives to an appropriately chosen subset of the data in order to see which was fastest. Several trials could be done. The results could then be used to "predict" which alternative would be fastest on the complete data set. Thus, the system could "learn" from samples to favor alternatives which perform better.

Finally, a complete system for concurrent execution of alternatives should be built. This thesis identifies the opportunities and provides enough of a roadmap so that the implementation should be straightforward; source code for the RB processor and *rfork()* are already available. The completed system should provide a chance to develop better execution strategies and further, refine the boundaries of the domain for which performance improvements are possible.

8. References

[Abramson1985a] D.A. Abramson and J.L. Keedy, "Implementing a large virtual memory in a distributed computing system," in *Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences* (1985), pp. 515-522.

[Anderson1967a] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling," *IBM Journal of Research and Development*, pp. 8-24 (January 1967).

[AT&T1986a] AT&T, *WE 32101 Memory Management Unit Information Manual*, Call 1-800-432-6600; Select Code 307-731, November 1986.

[Bach1986a] M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall (1986).

[Barak1985a] Amnon Barak and Amnon Shiloh, "A Distributed Load-balancing Policy for a Multicomputer," *SOFTWARE - PRACTICE AND EXPERIENCE* **15**(9), pp. 901-913 (September 1985).

[Bartlett1978a] J. F. Bartlett, "A NonStop Operating System," in *Proceedings, Eleventh Hawaii International Conference on on System Sciences* (1978), pp. 103-117.

[Bartlett1981a] J. F. Bartlett, "A NonStop Kernel," in *Proceedings, Eighth ACM Symposium on Operating Systems Principles* (1981), pp. 22-29.

[Beck1985a] Bob Beck and Bob Kasten, "VLSI Assist in Building a Multiprocessor UNIX System," in *USENIX Proceedings* (June 1985), pp. 255-275.

[Becker1984a] Richard A. Becker and John M. Chambers, *S - An Interactive Environment for Data Analysis and Graphics*, Wadsworth, 1984.

[Bernstein1980a] Arthur J. Bernstein, "Output Guards and Nondeterminism in "Communicating Sequential Processes"," *ACM Transactions on Programming Languages and Systems* 2(2), pp. 234-238 (April, 1980).

[Bobrow1972a] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10," *Communications of the ACM* **15**(3), pp. 135-143 (March 1972).

[Borg1983a] Anita Borg, J. Baumbach, and S. Glazer, "A message system supporting fault tolerance.," in *Proceedings, Ninth ACM Symposium on Operating Systems*

Principles (ACM Operating Systems Review) (1983), pp. 90-99.

[Bourne1978a] S.R. Bourne, "The UNIX Shell," *The Bell System Technical Journal* **57**(6, Part 2), pp. 1971-1990 (July-August 1978).

[BSD1982a] BSD, *UNIX User's Manual, 4.2 BSD*, University of California, Berkeley (1982).

[Butler1988a] Ralph Butler, Terry Disz, Ewing Lusk, Robert Olson, Ross Overbeek, and Rick Stevens, "Scheduling OR-Parallelism: An Argonne Perspective," in *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, ed. Robert A. Kowalski and Kenneth A. Bowen (1988), pp. 1590-1605. MIT Press

[Cha1987a] S. D. Cha, N. G. Leveson, T. J. Shimeall, and J. C. Knight, "An Empirical Study of Software Error Detection Using Self-Checks," in *Digest of Papers, The Seventeenth International Symposium on Fault-Tolerant Computing*, Pittsburgh, Pennsylvania (July 6-8, 1987), pp. 156-161.

[Chambers1983a] John M. Chambers, William S. Cleveland, Beat Kleiner, and Paul A. Tukey, *Graphical Methods for Data Analysis*, Duxbury Press, 1983.

[Chang1989a] Y. F. Chang, "Coefficients of wilk13," *Personal Communication* (February 9, 1989).

[Cheriton1986a] D. R. Cheriton, "Problem-oriented Shared Memory: A Decentralized Approach to Distributed System Design," in *Proceedings, Sixth International Conference on Distributed Computing Systems* (1986), pp. 190-197.

[Ciepelewski1985a] Andrzej Ciepelewski and Seif Haridi, "Execution of Bagof on the OR-Parallel Token Machine," in *Proceedings of the International Conference on Fifth Generation Computer Systems* (November 1985), pp. 551-562. ICOT

[Clark1987a] Keith Clark and Steve Gregory, "A Relational Language for Parallel Programimg," in *Concurrent Prolog (Collected Papers)*, ed. Ehud Shapiro (1987), pp. 9-26. MIT Press

[Clark1987b] Keith Clark and Steve Gregory, "PARLOG: Parallel Programming in Logic," in *Concurrent Prolog (Collected Papers)*, ed. Ehud Shapiro (1987), pp. 84-139. MIT Press

[Clocksin1984a] W. F. Clocksin and C. S. Mellish, *Programming in Prolog (2nd Edition)*, Springer-Verlag (1984).

[Cohen1979a] Jacques Cohen, "Non-Deterministic Algorithms," *ACM Computing Surveys* **11**(2), pp. 79-94 (June 1979).

[Cooper1984a] Eric Charles Cooper, "Circus: A replicated procedure call facility," in *Proceedings of the 4th Symposium on Reliability in Distributed Software and Database Systems* (October 1984), pp. 11-24.

[Cooper1985a] Eric Charles Cooper, "Replicated Distributed Programs," Ph.D. Thesis, University of California, Berkeley (1985).

[Crammond1985a] J. Crammond, "A Comparative Study of Unification Algorithms for OR-Parallel Execution of Logic Languages," *IEEE Transactions on Computers* **C-34**(10), pp. 911-917 (October 1985).

[Deitel1984a] H.M. Deitel, *An Introduction to Operating Systems (Revised First Edition),* Addison-Wesley (1984).

[Delp1988a] Gary Delp, Ravi Sethi, and David Farber, "An analysis of Memnet: An experiment in high-speed shared-memory local networking," in *Proceedings,SIGCOMM'88 Symposium*, Stanford (August, 1988).

[DeWitt1973a] Bryce DeWitt and R. Neill Graham, *The Many Worlds Interpretation of Quantum Mechanics*, Princeton University Press, 1973.

[Dijkstra1976a] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J. (1976).

[Disz1987a] Terry Disz, Ewing Lusk, and Ross Overbeek, "Experiments with OR-Parallel Logic Programs," in *Proceedings of the Fourth International Conference on Logic Programming*, ed. Jean-Louis Lassez (1987), pp. 576-600. MIT Press

[Encore1985a] Encore, *Multimax Multiprocessor System,* Encore Computer Corporation (1985). Sales Literature

[Eswaran1976a] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Communications of the ACM* **19**, pp. 624-633 (November 1976).

[Feller1970a] W. Feller, *An Introduction to Probability Theory and Its Applications*, Wiley, New York (1970).

[Ferguson1988a] Donald Ferguson, Yechiam Yemini, and Christos Nikolaou, "Microeconomic Algorithms for Load Balancing in Distributed Computer Systems,"

8th International Conference on Distributed Computing Systems, San Jose, CA, pp. 491-499, IEEE Computer Society (June 1988).

[Feynman1963a] Richard P. Feynman, Robert B. Leighton, and Matthew Sands, *The Feynman Lectures on Physics*, Addison-Wesley, Reading, MA (1963).

[Galambos1987a] J. Galambos, *The Asymptotic Theory of Extreme Order Statistics. 2nd Edition*, Krieger (1987).

[Garcia1989a] Armando Garcia, Richard Freitas, and David Foster, *The Advanced Computing Environment Multiprocessor Workstation*, IBM Research Division (March 22, 1989).

[Goldberg1987a] Arthur P. Goldberg and David R. Jefferson, "Transparent Process Cloning: A Tool for Load Management of Distributed Programs," in *Proceedings, International Conference on Parallel Processing* (1987), pp. 728-734.

[Gray1988a] Jim Gray, "Why do systems fail?," *Columbia Computer Science Department Lecture* (Fall 1988).

[Gray1978a] J. N. Gray, "Notes on Data Base Operating Systems," in *Operating Systems: An Advanced Course*, ed. G. Seegmueller (1978), pp. 393-481. Springer

[Gray1981a] J. N. Gray, "The transaction concept: Virtues and Limitations," in *Proceedings, VLDB Conference*, Cannes, France (September 1981).

[Hausman1987a] Bogumil Hausman, Andrzej Ciepielewski, and Seif Haridi, "OR-parallel Prolog Made Efficient on Shared Memory Multiprocessors," Tech. Report, Swedish Institute of Computer Science (1987).

[Hecht1979a] H. Hecht, "Fault-Tolerant Software," *IEEE Transactions on Reliability*, pp. 227-232 (August 1979).

[Horning1974a] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith, and B. Randell, "A program structure for error detection and recovery.," in *Proceedings, Conference on Operating Systems: Theoretical and Practical Aspects* (April 1974), pp. 177-193.

[Jefferson1987a] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLoreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H. Younger, and S. Bellenot, "Time Warp Operating System," *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, Austin, TX, pp. 77-93, In *ACM Operating Systems Review 21*:5 (8-11 November 1987).

[Jefferson1985a] David R. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems* **7**(3), pp. 404-425 (July, 1985).

[Jenkins1970a] M. A. Jenkins and J. F. Traub, "A Three-Stage Variable-Shift Iteration for Polynomial Zeros and its Relation to Generalized Rayleigh Iteration," *Numer. Math.* **14**, pp. 252-263 (1970).

[Jenkins1972a] M. A. Jenkins and J. F. Traub, "Algorithm 419: Zeros of a Complex Polynomial," *Communications of the ACM* **15**, pp. 97-99 (February, 1972).

[Johnson1987a] Thomas D. Johnson, Jonathan M. Smith, and Eric S. Wilson, "Disk Response Time Measurements," in *Proceedings, Winter 1987 USENIX Technical Conference*, Washington, DC (January, 1987), pp. 147-162.

[Joy1982a] W. Joy, *4.2BSD System Manual*, 1982.

[Kowalski1979a] Robert Kowalski, *Logic for Problem Solving*, North-Holland (1979).

[Kruskal1984a] V. Kruskal, "Managing Multi-version Programs with an Editor," *IBM Journal of Research and Development* **28**(1), pp. 74-81 (January, 1984).

[Kung1981a] H. T. Kung and John T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems* **6**(2), pp. 213-226 (June, 1981).

[Lampson1981a] Butler W. Lampson, "Atomic Transactions," in *Distributed Systems — Architecture and Implementation (An Advanced Course)*, ed. H. J. Siegel, Spring-Verlag (1981), pp. 246-265.

[Leach1982a] P.J. Leach, B.L. Stumpf, J.A. Hamilton, and P.H. Levine, "UID's as internal names in a distributed file system," in *Proceedings of the 1st Symposium on Principles of Distributed Computing* (1982), pp. 34-41.

[Leach1983a] P.J. Leach, P.H. Levine, B.P. Douros, J.A. Hamilton, D.L. Nelson, and B.L. Stumpf, "The architecture of an integrated local network," *IEEE Journal Selected Areas of Communication*, pp. 842-856 (1983).

[Ledgard1981a] Henry F. Ledgard, *Ada, An Introduction*, Springer-Verlag (1981). in same volume: Ada Reference Manual, (July 1980)

[Leffler1984a] Samuel J. Leffler, Michael J. Karels, and Marshall Kirk McKusick, "Measuring and Improving the Performance of 4.2BSD," in *Proceedings, Summer 1984 USENIX Technical Conference*, Salt Lake City, Utah (June 12-15, 1984),

pp. 237-252.

[Leland1986a] Will E. Leland and Teunis J. Ott, ''Load-balancing Heuristics and Process Behavior,'' in *Proceedings, ACM SigMetrics Performance 1986 Conference* (1986).

[Lelewer1987a] Debra A. Lelewer and Daniel S. Hirschberg, ''Data Compression,'' *ACM Computing Surveys* **19**(3), pp. 261-296 (September 1987).

[Li1986a] Kai Li and Paul Hudak, ''Memory Coherence in Shared Virtual Memory Systems,'' in *Proceedings, 5th ACM Symposium on Principles of Distributed Computing* (August, 1986), pp. 229-239.

[Li1986b] Kai Li, ''Shared Virtual Memory on Loosely Coupled Multiprocessors,'' in *Proceedings, International Conference on Computer Languages*, Miami, FL (October, 1986), pp. 98-106.

[Lilja1988a] David J. Lilja, ''Reducing the Branch Penalty in Pipelined Processors,'' *IEEE Computer* **21**(7), pp. 47-55 (July 1988).

[Lindstrom1984a] Gary Lindstrom and Prakash Panangaden, ''Stream based execution of logic programs,'' in *Proceedings, International Symposium on Logic Programming*, IEEE, Atlantic City, NJ (February, 1984), pp. 168-176.

[Lindstrom1984b] Gary Lindstrom, ''OR-Parallelism on Applicative Architectures,'' in *Proceedings, Second International Logic Programming Conference*, Uppsala University, Uppsala, Sweden (July 2-6, 1984), pp. 159-170.

[Lusk1988a] Ewing Lusk, Ralph Butler, Terrence Disz, Robert Olson, Ross Overbeek, Rick Stevens, David H. D. Warren, Alan Calderwood, Peter Szeredi, Seif Haridi, Per Brand, Mats Carlsson, Andrzej Ciepielewski, and Bogumil Hausman, ''The Aurora Or-Parallel Prolog System,'' Technical Report, Swedish Institute of Computer Science (April 29, 1988).

[Maguire,Jr.1988a] G. Q. Maguire,Jr., ''PACS for those interested in image processing: an expert configuration system,'' in *Les Entretiens de Lyon - Computer Science and Life: Medical Imaging and Experts Systems Applied to Medicine* (1988). S.E.E., C.E.R.F., and S.F.B.M.N.

[McKusick1985a] Marshall Kirk McKusick, Michael J. Karels, and Samuel J. Leffler, ''Performance Improvements and Functional Enhancements in 4.3BSD,'' in

Proceedings, Summer 1985 USENIX Technical Conference, Portland, Oregon (June 11-14, 1985), pp. 519-532.

[Nelson1987a] David Nelson, *Network Computing (videotaped lecture)*, University Video Communications, Stanford, CA (1987). P.O. Box 2666

[Nelson1984a] D.L. Nelson and P.J. Leach, "The Architecture and Applications of the Apollo Domain," *IEEE Computer Graphics*, pp. 58-66 (April 1984).

[Nelson1988a] M. Nelson, B. Welch, and J. Ousterhout, "Caching in the Sprite Network File System," *ACM Transactions on Computer Systems* **6**(1), pp. 134-154, Originally presented at the *Eleventh ACM Symposium on Operating Systems Principles* (February, 1988).

[Nguyen1988a] T.M. Nguyen, V.P. Srini, and A.M. Despain, "A Two-Tier Memory Architecture for High Performance Multiprocessor Systems," *Proceedings of the 1988 ACM International Conference on Supercomputing*, Saint-Malo, France (July 1988).

[Nilsson1980a] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing Company, 1980.

[Organick1972a] Elliott I. Organick, *The Multics System*, Massachusetts Institute of Technology Press (1972).

[Poplawski1987a] D.A. Poplawski and D.O. Rich, "Code Paging on Hypercubes," in *Proceedings of the 1987 International Conference on Parrallel Processing* (August 17-21, 1987). Pennsylvania State University Press

[Ralston1978a] Anthony Ralston and Philip Rabinowitz, *A First Course in Numerical Analysis (2nd Edition)*, McGraw-Hill, 1978.

[Randell1975a] B. Randell, "System structure for software fault tolerance," *IEEE Transactions on Software Engineering* **SE-1**, pp. 220-232 (June 1975).

[Reed1978a] David P. Reed, "Naming and Synchronization in a Decentralized Computer System," Technical Report 205 (Ph.D. Thesis) (September, 1978). MIT LCS

[Rice1968a] J. R. Rice, "On the Construction of Polyalgorithms for Automatic Numerical Analysis," in *Interactive Systems for Experimental Applied Mathematics*, ed. J. Reinfelds (1968), pp. 301-313.

[Rice1969a] John R. Rice, "A Polyalgorithm for the Automatic Solution of Nonlinear Equations," in *Proceedings, ACM National Conference* (1969), pp. 179-183.

[Rice1973a] John R. Rice, "NAPSS-like systems: Problems and Prospects," in *Proceedings, National Computer Conference* (1973), pp. 43-47.

[Rice1988a] John R. Rice, *Private Communication on NAPSS*, October, 1988.

[Rich1983a] Elaine Rich, *Artificial Intelligence,* McGraw-Hill (1983).

[Riseman1972a] Edward M. Riseman and Caxton C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps," *IEEE Transactions on Computers* **C-21**(12), pp. 1405-1411 (December 1972).

[Ritchie1978a] D.M. Ritchie and K.L. Thompson, "The UNIX Time-Sharing System," *Bell System Technical Journal* **57**(6), pp. 1905-1930 (July-August 1978).

[Robbins1975a] Herbert Robbins and John Van Ryzin, *Introduction to Statistics*, SRA (1975).

[Rochkind1975a] M. J. Rochkind, "The Source Code Control System," *IEEE Transactions on Software Engineering* **SE-1**, pp. 364-370 (1975).

[Ross1986a] M. L. Ross and K. Ramamohanarao, "Paging Strategy for Prolog Based Dynamic Virtual Memory," in *IEEE Symposium on Logic Programming*, Salt Lake City, UT (1986), pp. 46-57.

[Russell1978a] Richard M. Russell, "The Cray-1 Computer System," *Communications of the ACM* **21**(1), pp. 63-72 (January 1978).

[Sacerdoti1974a] E. D. Sacerdoti, "Planning in a Hierarchy of Abstraction Spaces," *Artificial Intelligence* **5** (1974).

[Sandberg1985a] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and R. Lyon, "The Design and Implementation of the Sun Network File System," in *USENIX Proceedings* (June 1985), pp. 119-130.

[Schneider1982a] Fred B. Schneider, "Synchronization in Distributed Programs," *ACM Transactions on Programming Languages and Systems* 4(2), pp. 179-195 (April 1982).

[Schneider1983a] F. B. Schneider and Richard D. Schlichting, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Transactions on Computer Systems* **1**(3), pp. 222-238 (August 1983).

[Shapiro1986a] Ehud Shapiro, "Concurrent Prolog: A Progress Report," *IEEE Computer* **19**(8), pp. 44-58 (August 1986).

[Smith1982a] A. J. Smith, "Cache Memories," *ACM Computing Surveys* **14**(3), pp. 473-530 (September 1982).

[Smith1986a] Jonathan M. Smith, "Approaches to Distributed UNIX Systems," Technical Report CUCS-223-86, Columbia University Computer Science Department (1986).

[Smith1988b] Jonathan M. Smith, "A Survey of Process Migration Mechanisms," *ACM SIGOPS Operating Systems Review*, pp. 28-40 (July, 1988).

[Smith1988c] Jonathan M. Smith and Gerald Q. Maguire,Jr., "The RB Language," Technical Report Number CUCS-364-88, Columbia University Computer Science Department (1988).

[Smith1988a] Jonathan M. Smith and Gerald Q. Maguire,Jr., "Effects of copy-on-write memory management on the response time of UNIX *fork* operations," *Computing Systems* **1**(3), pp. 255-278 (1988).

[Smith1989a] Jonathan M. Smith and John Ioannidis, "Implementing remote *fork()* with checkpoint/restart," *IEEE Technical Committee on Operating Systems Newsletter*, pp. 12-16 (February, 1989).

[Stallman1986a] Richard Stallman, *GNU Emacs Manual, Fourth Edition, Version 17*, Free Software Foundation, Inc., 100 Mass Ave., Cambridge, MA 02138 (February 1986).

[Stout1983a] Q. F. Stout, "Mesh-Connected Computers with Broadcasting," *IEEE Transactions on Computers* **32**(9), pp. 826-830 (September 1983).

[Strom1985a] R. E. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Transactions on Computer Systems* 3(3), pp. 204-226 (August 1985).

[Strom1987a] R. E. Strom and S. Yemini, "Synthesizing Distributed and Parallel Programs through Optimistic Transformations," in *Current Advances in Distributed Computing and Communications* (1987). Computer Science Press

[Symes1971a] L. R. Symes, "Evaluation of NAPSS Expressions Involving Polyalgorithms, Functions, Recursion, and Untyped Variables," in *Mathematical Software*, ed. John R. Rice, Academic Press (1971), pp. 261-274. (Based on the proceedings of the Mathematical Software Symposium held at Purdue University, Lafayette, Indiana, April 1-3, 1970)

[Theimer1985a] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton,

"Preemptable Remote Execution Facilities for the V-System," in *Proceedings, 10th ACM Symposium on Operating Systems Principles* (1985), pp. 2-12.

[Thomas1979a] R. H. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Transactions on Database Systems* **4**(2), pp. 180-209 (June 1979).

[Thornton1970a] J. E. Thornton, *Design of a Computer: The CDC 6600,* Scott, Foresman & Co., Glenview, IL (1970).

[Tichy1985a] W. Tichy, "RCS - A System for Version Control," *Software - Practice and Experience* **15**(7), pp. 637-654 (July, 1985).

[Tick1988a] Evan Tick, *Memory Performance of Prolog Architectures*, Kluwer, 1988.

[Tinker1987a] Peter Allmond Tinker, "The Design and Implementation of an OR-Parallel Logic Programming System," Ph.D. Thesis, University of Utah (August 1987).

[Traiger1982a] I.L. Traiger, J. Gray, C.A. Galtieri, and B.G. Lindsay, "Transactions and Consistency in Distributed Database Systems," *ACM Transactions on Database Systems* 7(3), pp. 323-342 (September 1982).

[Traub1964a] J. F. Traub, *Iterative Methods for the Solution of Equations,* Prentice-Hall, Englewood Cliffs, NJ (1964). Appendix F

[Trotter1941a] W. Trotter, *Collected Papers of Wilfred Trotter*, Oxford University Press, 1941.

[Ueda1987a] Kazunori Ueda, "Guarded Horn Clauses," in *Concurrent Prolog (Collected Papers)*, ed. Ehud Shapiro (1987), pp. 140-156. MIT Press

[Warren1987a] David H. D. Warren, "Or-Parallel Execution Models of Prolog," in *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, ed. H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, Pisa, Italy (March 1987), pp. 243-259. Springer-Verlag

[Weinberger1984a] P.J. Weinberger, "The Version 8 Network File System," in *USENIX Proceedings* (June 1984), p. 86.

[Wise1986a] Michael J. Wise, *PROLOG Multiprocessors*, Prentice-Hall, 1986.

[Young1987a] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proceedings of the Eleventh*

ACM Symposium on Operating Systems Principles, Austin, TX, pp. 63-76. In *ACM Operating Systems Review 21*:5 (8-11 November 1987).

[Zayas1987a] E. Zayas, ''Attacking the Process Migration Bottleneck,'' *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, Austin, TX, pp. 13-24, In *ACM Operating Systems Review 21*:5 (8-11 November 1987).

9. Appendix I: *do_fork.c*

```
#include <errno.h>
#include <sys/param.h>

#ifndef NBPC
#define PAGE_SIZE 2048
#else
#define PAGE_SIZE NBPC
#endif

#ifndef NULL
#define NULL 0
#endif

main( argc, argv )
int argc;
char *argv[];
{
   int count = 0, heap_size = 0, pid, status;
   double atof(), write_fraction = 0.0,
      write_count = 0.0, write_size = 0.0;
   register char *ptr;
   char *malloc();
   extern int errno;

   if( argc > 1 )
   {
      count = atoi( argv[1] );
      if( argc > 2 )
      {
         heap_size = atoi( argv[2] );
         if( ( ptr = malloc( heap_size ) )
            == (char *) NULL )
            error( "Insufficient memory available. Exiting.\n" );
         if( argc > 3 )
         {
            write_fraction = atof( argv[3] );
            if( write_fraction < 0.0 || write_fraction > 1.0 )
               error( "0.0 <= writes <= 1.0; Exiting.\n" );
            write_size = write_fraction * (double) heap_size;
         }
      }
   }

   while( count > 0 )
   {
      switch( (pid = fork()) )
      {
      case -1:   /* failed. If EAGAIN, wait. */
         if( errno == EAGAIN )
            wait( &status );
         break;
```

```
        case 0:       /* child. make refs if needed, and exit */
            while( write_count < write_size )
            {
                *ptr = ' ';
                ptr = &ptr[PAGE_SIZE];
                write_count += (double) PAGE_SIZE;
            }
            exit( 0 );

        default:
            count -= 1;
        }
    }

    exit( 0 );
}

error( string )
char *string;
{
    write( 2, string, strlen( string ) );
    exit( 1 );
}
```

10. Appendix II: netrand.c

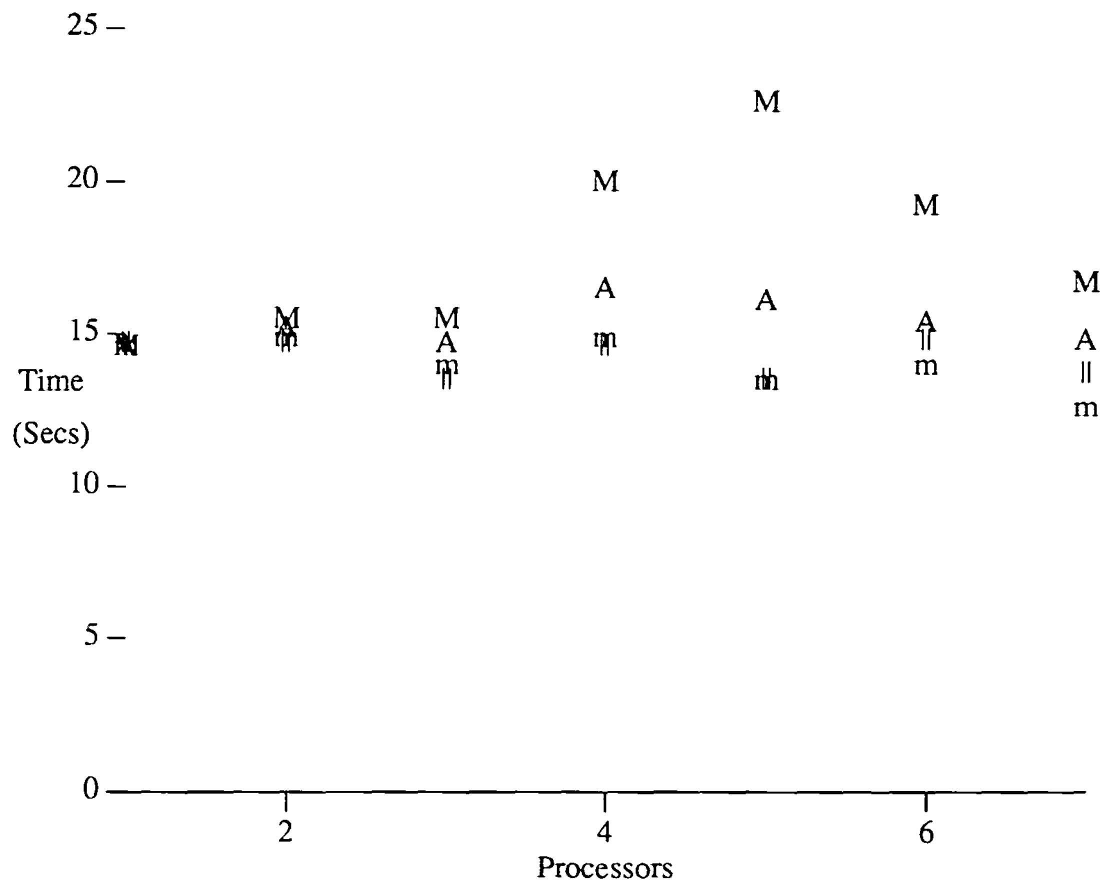
```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/times.h>

long times();
struct tms tbuf;

#include <stdio.h>

main( argc, argv )
int argc;
char *argv[];
{
   long start, diff;
   int count;
   FILE *fp;

   if( argc < 2 )
   {
      fprintf( stderr, "number required.\n" );
      exit( 1 );
   }
   count = atoi( argv[1] );
   if( count <= 0 )
      exit( 1 );
   if( argc > 2 )
   {
      fp = fopen( argv[2], "r" );
      if( fp == (FILE *) NULL )
      {
         perror( argv[1] );
         exit( 1 );
      }
   }
   else
      fp = stdin;

   start = times( &tbuf );

   random_read( fp, count );

   diff = times( &tbuf ) - start;
   printf( "elapsed time, %g seconds.\n", (float) diff / (float) HZ );
   exit( 0 );

}

random_read( fp, count )
FILE *fp;
int count;
{
```

```
   int i, fd, block_no, bytes;
   char page[NBPG_M320];
#ifdef DEBUG
   FILE *lfp;

   lfp = fopen( "/tmp/RANDBLOCKS", "w" );
   if( lfp == (FILE *) NULL )
      lfp = stdout;
#endif

   fd = fileno( fp );

   for( i = 0, block_no = random( times( &tbuf ) ) % count;
      i < count;
      block_no = random( 0 ) % count, i = i + 1 )
   {
      lseek( fd, block_no * sizeof(page), 0 );
      bytes = read( fd, page, sizeof(page));
#ifdef DEBUG
      fprintf( lfp, "block_no: %d, bytes read: %d\n",
         block_no, bytes );
   }
   fclose( lfp );
#else
   }
#endif

   return;
}

int
random( start )
int start;
{
   static int last_val;

   if( start != 0 )
      last_val = start;

   last_val = (17*last_val+123) % 65521;
   while( last_val < 0 )
      last_val += 65521;

   return( last_val );
}
```

11. Appendix III: Further Jenkins-Traub executions

11.1. Polynomial #1

Degree 43 polynomial: z^{43}+ $(1.890-0.19491576i)\cdot z^{42}$+ $(-0.140045104-0.161369678i)\cdot z^{41}$+ $(-0.1274642064-0.381680467i)\cdot z^{40}$+ $(-0.743821643-0.5810837248i)\cdot z^{39}$+ $(-0.16752870876-0.10043831986i)\cdot z^{38}$+ $(1.41763432806+0.29981366628i)\cdot z^{37}$+ $(1.23799211445-0.106982985377i)\cdot z^{36}$+ $(-0.189267174496-0.344079321 81i)\cdot z^{35}$+ $(-0.152123766242+0.236186810121i)\cdot z^{34}$+ $(-0.198013830239+0.280742352179i)\cdot z^{33}$+ $(-0.342742413912+0.83786997252i)\cdot z^{32}$+ $(-0.37460901644-0.301901877085i)\cdot z^{31}$+ $(-0.195080181222-0.100059575279i)\cdot z^{30}$+ $(-0.94732085290-0.90223058904i)\cdot z^{29}$+ $(1.27282343436-0.57440498715i)\cdot z^{28}$+ $(1.24271198674-0.3724963179i)\cdot z^{27}$+ $(1.682559499-0.7237864131i)\cdot z^{26}$+ $(-0.1528494152-0.397912167i)\cdot z^{25}$+ $(1.63349391+0.237887291i)\cdot z^{24}$+ $(1.27878345+0.0897986i)\cdot z^{23}$+ $(1.0802320-0.1101871i)\cdot z^{22}$+ $(-0.10+0.0i)\cdot z^{21}$+ $(1.890-0.19491576i)\cdot z^{20}$+ $(-0.140045104-0.161369678i)\cdot z^{19}$+ $(-0.1274642064-0.381680467i)\cdot z^{18}$+ $(-0.743821643-0.5810837248i)\cdot z^{17}$+ $(-0.16752870876-0.10043831986i)\cdot z^{16}$+ $(1.41763432806+0.29981366628i)\cdot z^{15}$+ $(1.23799211445-0.106982985377i)\cdot z^{14}$+ $(-0.189267174496-0.34407932181i)\cdot z^{13}$+ $(-0.152123766242+0.236186810121i)\cdot z^{12}$+ $(-0.198013830239+0.280742352179i)\cdot z^{11}$+ $(-0.342742413912+0.83786997252i)\cdot z^{10}$+ $(-0.37460901644-0.301901877085i)\cdot z^{9}$+ $(-0.195080181222-0.100059575279i)\cdot z^{8}$+ $(-0.94732085290-0.90223058904i)\cdot z^{7}$+ $(1.27282343436-0.57440498715i)\cdot z^{6}$+ $(1.24271198674-0.3724963179i)\cdot z^{5}$+ $(1.682559499-0.7237864131i)\cdot z^{4}$+ $(-0.1528494152-0.397912167i)\cdot z^{3}$+ $(1.63349391+0.237887291i)\cdot z^{2}$+ $(1.27878345+0.0897986i)\cdot z^{1}$+ $(1.0802320-0.1101871i)$



11.2. Polynomial #2

Degree 43 Polynomial:

$z^{43}+ \quad (1.0e\,1.890+1.0e\,{-0.19491576}i)\cdot z^{42}+$

$(1.0e\,{-0.140045104}+1.0e\,{-0.161369678}i)\cdot z^{41}+$

$(1.0e\,{-0.1274642064}+1.0e\,{-0.381680467}i)\cdot z^{40}+$

$(1.0e\,{-0.743821643}+1.0e\,{-0.5810837248}i)\cdot z^{39}+$

$(1.0e\,{-0.16752870876}+1.0e\,{-0.10043831986}i)\cdot z^{38}+$

$(1.0e\,1.41763432806+1.0e\,1.29981366628i)\cdot z^{37}+$

$(1.0e\,1.23799211445+1.0e\,{-0.106982985377}i)\cdot z^{36}+$

$(1.0e\,{-0.189267174496}+1.0e\,{-0.3440793218}1i)\cdot z^{35}+$

$(1.0e\,{-0.152123766242}+1.0e\,1.236186810121i)\cdot z^{34}+$

$(1.0e{-0.198013830239}+1.0e\,1.280742352179i)\cdot z^{33}+$

$(1.0e{-0.342742413912}+1.0e\,1.83786997252i)\cdot z^{32}+$

$(1.0e{-0.37460901644}+1.0e{-0.301901877085}i)\cdot z^{31}+$

$(1.0e{-0.195080181222}+1.0e{-0.100059575279}i)\cdot z^{30}+$

$(1.0e{-0.94732085290}+1.0e{-0.90223058904}i)\cdot z^{29}+$

$(1.0e\,1.27282343436+1.0e{-0.57440498715}i)\cdot z^{28}+$

$(1.0e\,1.24271198674+1.0e{-0.3724963179}i)\cdot z^{27}+$

$(1.0e\,1.682559499+1.0e{-0.7237864131}i)\cdot z^{26}+$

$(1.0e{-0.1528494152}+1.0e{-0.397912167}i)\cdot z^{25}+$

$(1.0e\,1.63349391+1.0e\,1.237887291i)\cdot z^{24}+$ $(1.0e\,1.27878345+1.0e\,1.0897986i)\cdot z^{23}+$

$(1.0e\,1.0802320+1.0e{-0.1101871}i)\cdot z^{22}+$ $(1.0e{-0.10}+1.0e\,1.0i)\cdot z^{21}+$

$(1.0e\,1.890+1.0e{-0.19491576}i)\cdot z^{20}+$ $(1.0e{-0.140045104}+1.0e{-0.161369678}i)\cdot z^{19}+$

$(1.0e{-0.1274642064}+1.0e{-0.381680467}i)\cdot z^{18}+$

$(1.0e{-0.743821643}+1.0e{-0.5810837248}i)\cdot z^{17}+$

$(1.0e{-0.16752870876}+1.0e{-0.10043831986}i)\cdot z^{16}+$

$(1.0e\,1.41763432806+1.0e\,1.29981366628i)\cdot z^{15}+$

$(1.0e\,1.23799211445+1.0e{-0.106982985377}i)\cdot z^{14}+$

$(1.0e{-0.189267174496}+1.0e{-0.34407932181}i)\cdot z^{13}+$

$(1.0e{-0.152123766242}+1.0e\,1.236186810121i)\cdot z^{12}+$

$(1.0e{-0.198013830239}+1.0e\,1.280742352179i)\cdot z^{11}+$

$(1.0e{-0.342742413912}+1.0e\,1.83786997252i)\cdot z^{10}+$

$(1.0e{-0.37460901644}+1.0e{-0.301901877085}i)\cdot z^{9}+$

$(1.0e{-0.195080181222}+1.0e{-0.100059575279}i)\cdot z^{8}+$

$(1.0e{-0.94732085290}+1.0e{-0.90223058904}i)\cdot z^{7}+$

$(1.0e\,1.27282343436+1.0e{-0.57440498715}i)\cdot z^{6}+$

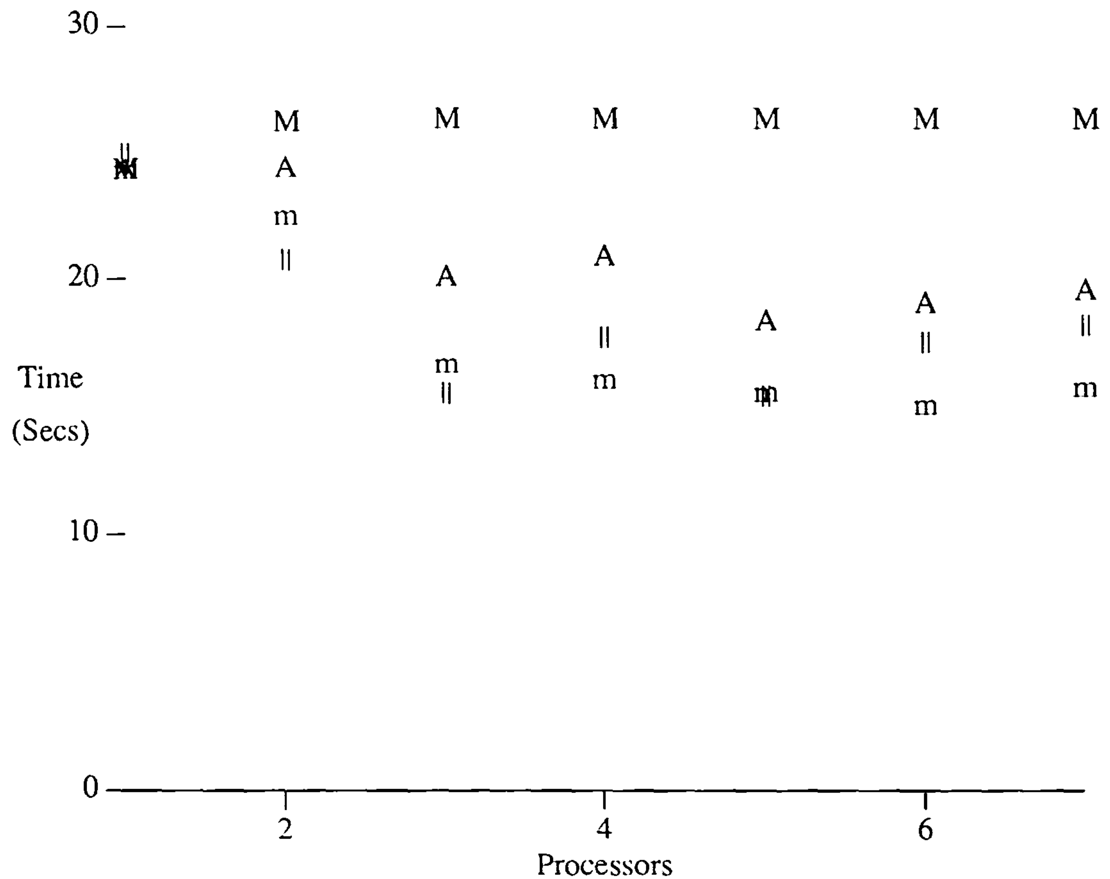
$(1.0e\,1.24271198674+1.0e{-0.3724963179}i)\cdot z^{5}+$

$(1.0e\,1.682559499+1.0e{-0.7237864131}i)\cdot z^{4}+$

$(1.0e{-0.1528494152}+1.0e{-0.397912167}i)\cdot z^{3}+$

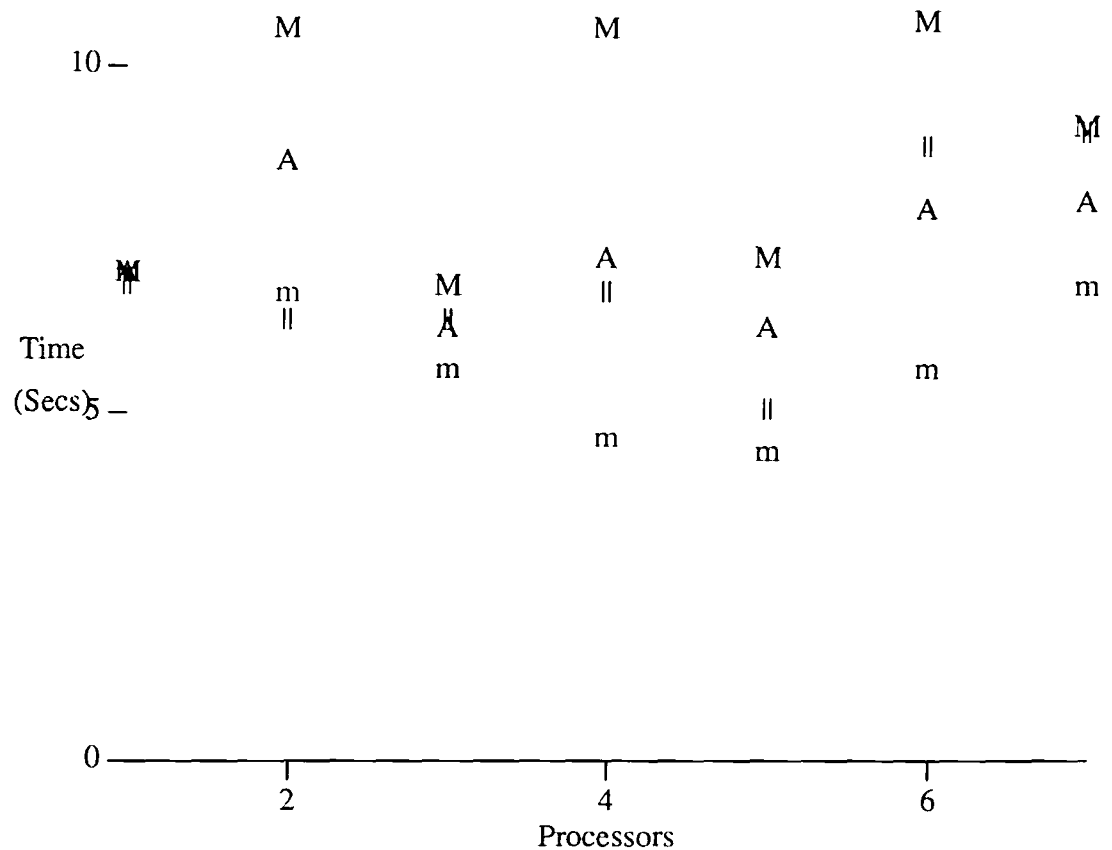
$(1.0e\,1.63349391+1.0e\,1.237887291i)\cdot z^{2}+$ $(1.0e\,1.27878345+1.0e\,1.0897986i)\cdot z^{1}+$

$(1.0e\,1.0802320+1.0e{-0.1101871}i)$



11.3. Polynomial #3

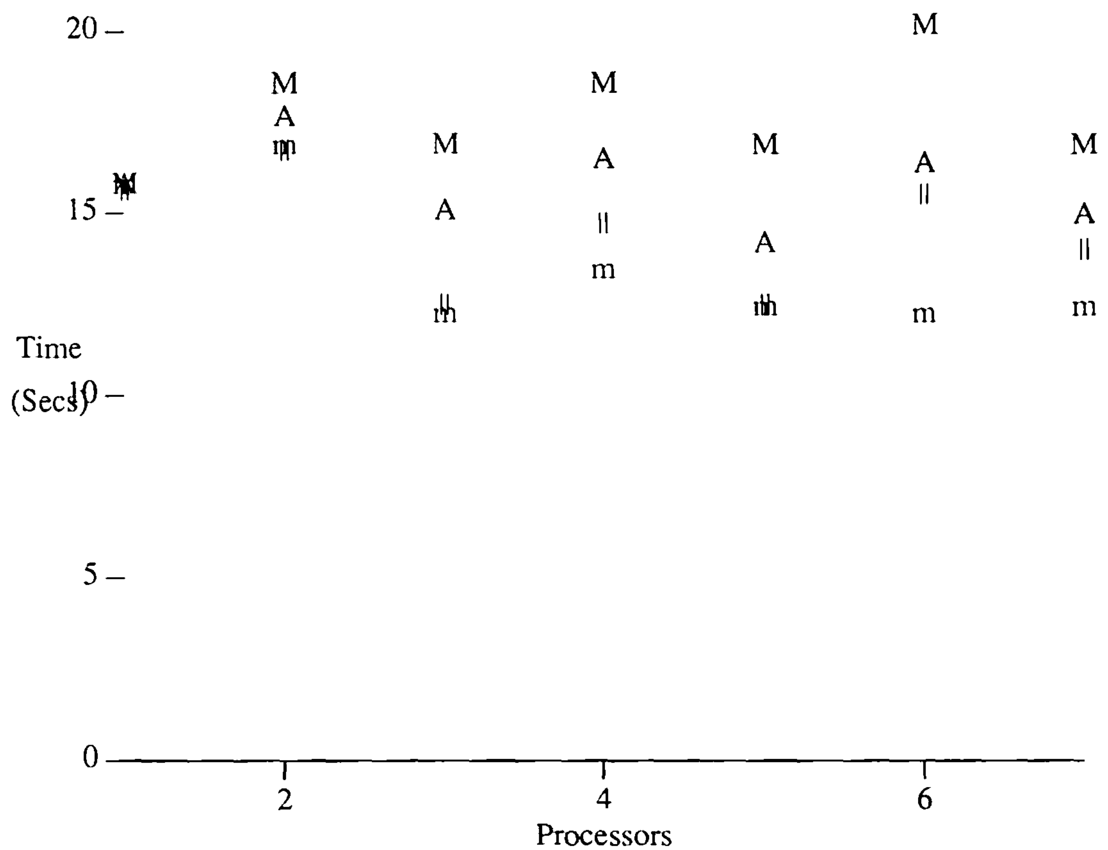
Degree 16 Polynomial: z^{16}+ $(199.80+33.20i)\cdot z^{15}$+ $(1766.930+588.940i)\cdot z^{14}$+ $(9400.2460+4944.7240i)\cdot z^{13}$+ $(32844.87470+24955.1830i)\cdot z^{12}$+ $(78211.15150+83755.778560i)\cdot z^{11}$+ $(125211.112583+194911.045162i)\cdot z^{10}$+ $(120455.424807+314055.685381i)\cdot z^{9}$+ $(27855.185034+327400.983162i)\cdot z^{8}$+ $(103788.708048+151722.462694i)\cdot z^{7}$+ $(168700.986162+135544.317038i)\cdot z^{6}$+ $(124300.788642+335711.308326i)\cdot z^{5}$+ $(36733.995257+333899.404113i)\cdot z^{4}$+ $(13600.752190+201800.264386i)\cdot z^{3}$+ $(16899.471245+76744.191752i)\cdot z^{2}$+ $(6222.122192+17000.277923i)\cdot z^{1}$+ $(866.407858+1688.238421i)$



11.4. Polynomial #4

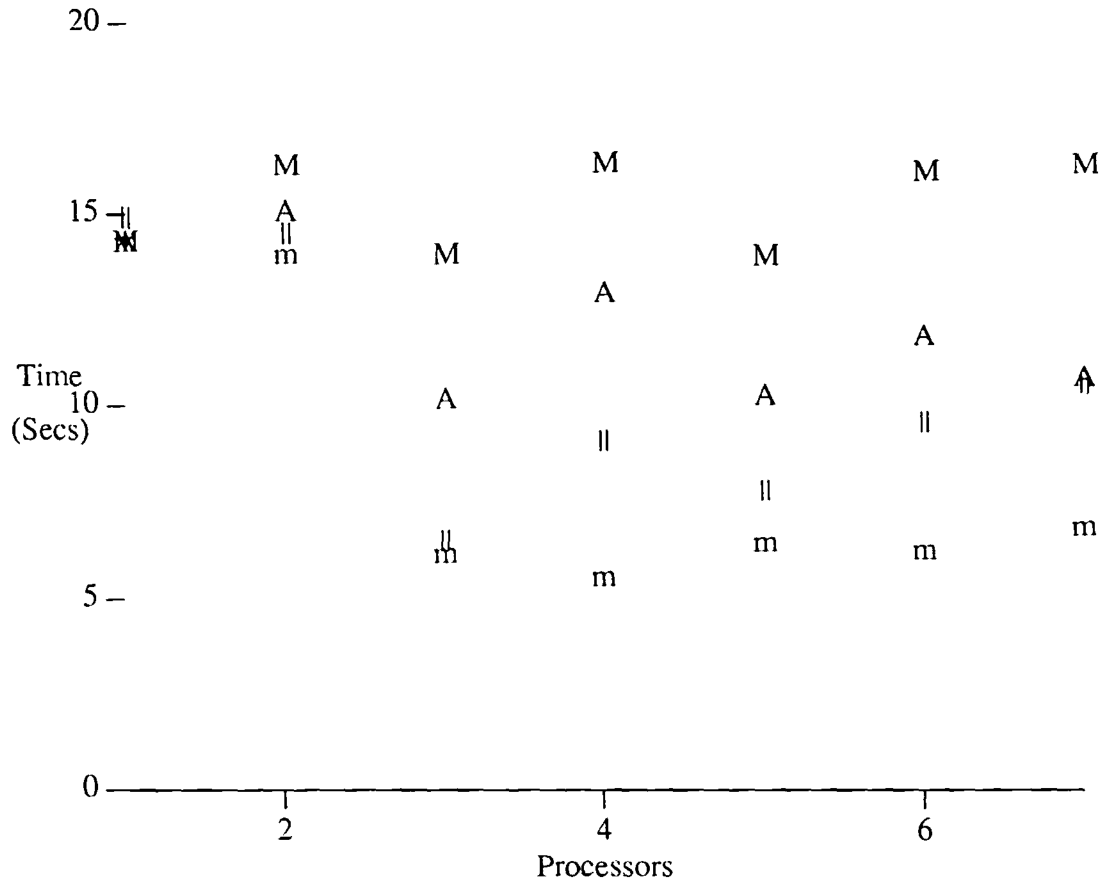
Degree 43 Polynomial $z^{43}+$ $(1.8900000-0.19491576i)\cdot z^{42}+$ $(-0.140045104-0.161369678i)\cdot z^{41}+$ $(-0.1274642064-0.381680467i)\cdot z^{40}+$ $(-0.743821643-0.5810837248i)\cdot z^{39}+$ $(-0.16752870876-0.10043831986i)\cdot z^{38}+$ $(1.41763432806+1.29981366628i)\cdot z^{37}+$ $(1.23799211445-0.106982985377i)\cdot z^{36}+$ $(-0.189267174496-0.344079321181i)\cdot z^{35}+$ $(-0.152123766242+1.236186810121i)\cdot z^{34}+$ $(-0.198013830239+1.280742352179i)\cdot z^{33}+$ $(-0.342742413912+1.83786997252i)\cdot z^{32}+$ $(-0.37460901644-0.301901877085i)\cdot z^{31}+$ $(-0.195080181222-0.100059575279i)\cdot z^{30}+$ $(-0.94732085290-0.90223058904i)\cdot z^{29}+$ $(1.27282343436-0.57440498715i)\cdot z^{28}+$ $(1.24271198674-0.3724963179i)\cdot z^{27}+$ $(1.682559499-0.7237864131i)\cdot z^{26}+$ $(-0.1528494152-0.397912167i)\cdot z^{25}+$ $(1.63349391+1.237887291i)\cdot z^{24}+$ $(1.27878345+1.0897986i)\cdot z^{23}+$ $(1.0802320-0.1101871i)\cdot z^{22}+$ $(-0.10+1.0i)\cdot z^{21}+$

$(1.890-0.19491576i)\cdot z^{20}+$ $(-0.140045104-0.161369678i)\cdot z^{19}+$
$(-0.1274642064-0.381680467i)\cdot z^{18}+$ $(-0.743821643-0.5810837248i)\cdot z^{17}+$
$(-0.16752870876-0.10043831986i)\cdot z^{16}+$ $(1.41763432806+1.29981366628i)\cdot z^{15}+$
$(1.23799211445-0.106982985377i)\cdot z^{14}+$ $(-0.189267174496-0.34407932181i)\cdot z^{13}+$
$(-0.152123766242+1.236186810121i)\cdot z^{12}+$ $(-0.198013830239+1.280742352179i)\cdot z^{11}+$
$(-0.342742413912+1.83786997252i)\cdot z^{10}+$ $(-0.37460901644-0.301901877085i)\cdot z^{9}+$
$(-0.195080181222-0.100059575279i)\cdot z^{8}+$ $(-0.94732085290-0.90223058904i)\cdot z^{7}+$
$(1.27282343436-0.57440498715i)\cdot z^{6}+$ $(1.24271198674-0.3724963179i)\cdot z^{5}+$
$(1.682559499-0.7237864131i)\cdot z^{4}+$ $(-0.1528494152-0.397912167i)\cdot z^{3}+$
$(1.63349391+1.237887291i)\cdot z^{2}+$ $(1.27878345+1.0897986i)\cdot z^{1}+$
$(1.0802320-0.1101871i)$



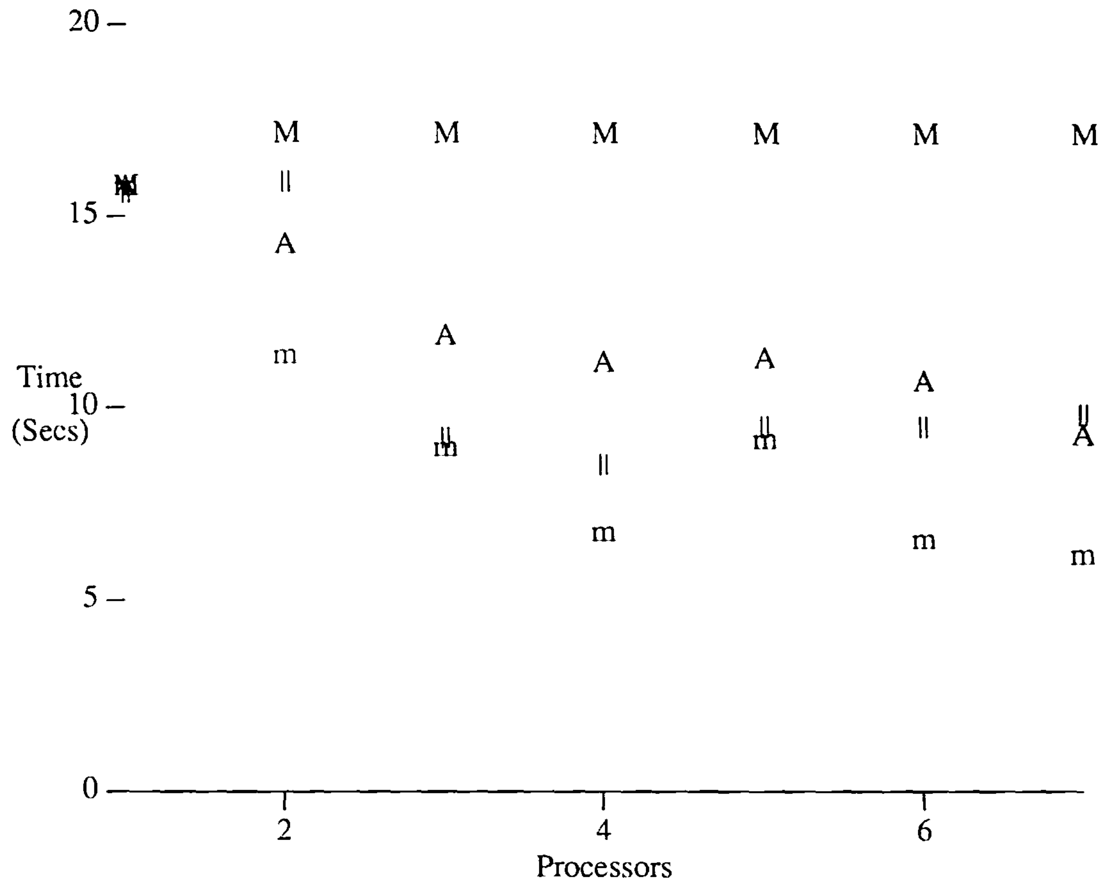
11.5. Polynomial #5

Degree 16 Polynomial: $z^{16}+$ $(1998.0+332.0i)\cdot z^{16}+$ $(17669.30+5889.40i)\cdot z^{14}+$ $(94002.460+49447.240i)\cdot z^{13}+$ $(328448.7470+249551.830i)\cdot z^{12}+$ $(782111.5150+837557.78560i)\cdot z^{11}+$ $(1252111.12583+1949110.45162i)\cdot z^{10}+$ $(1204554.24807+3140556.85381i)\cdot z^{9}+$ $(278551.85034+3274009.83162i)\cdot z^{8}+$ $(1037887.08048+1517224.62694i)\cdot z^{7}+$ $(1687009.86162+1355443.17038i)\cdot z^{6}+$ $(1243007.88642+3357113.08326i)\cdot z^{5}+$ $(367339.95257+3338994.04113i)\cdot z^{4}+$ $(136007.52190+2018002.64386i)\cdot z^{3}+$ $(168994.71245+767441.91752i)\cdot z^{2}+$ $(62221.22192+170002.77923i)\cdot z^{1}+$ $(8664.07858+16882.38421i)$



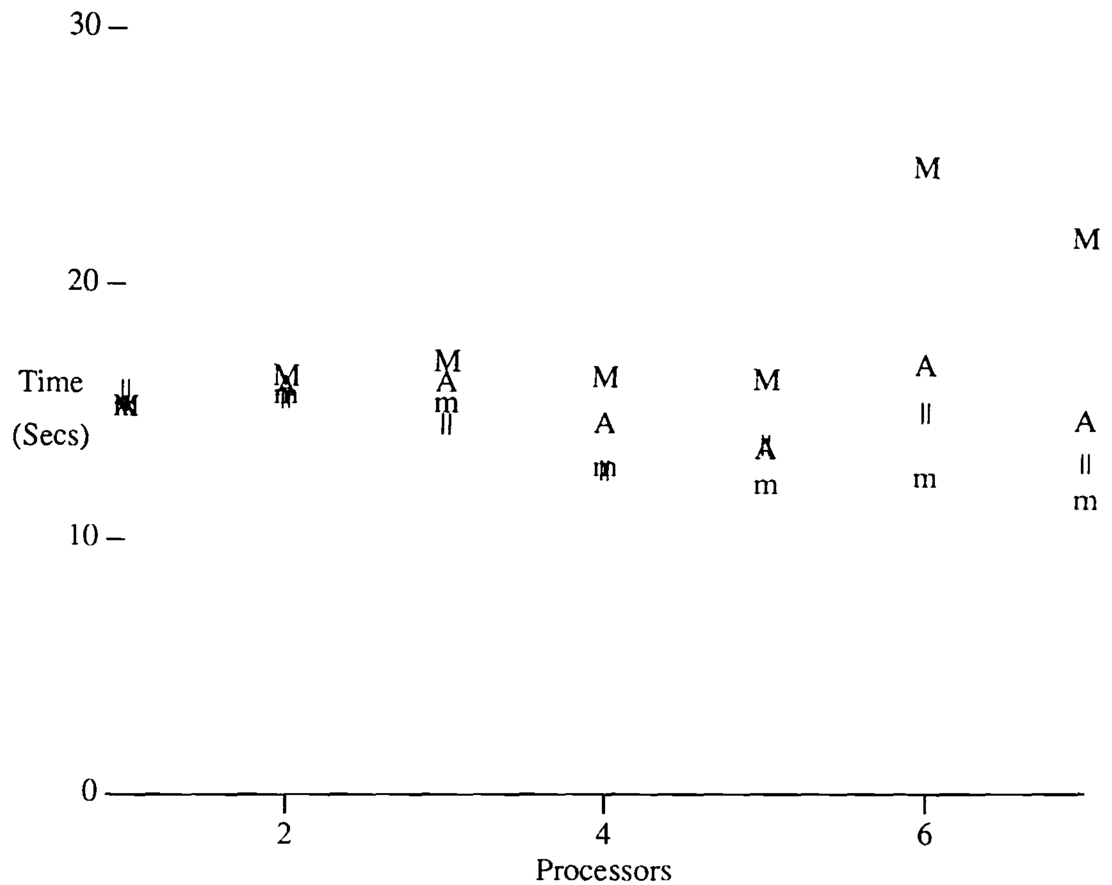
11.6. Polynomial #6

Degree 21 Polynomial: $z^{21}+(8.90+19.491576i)\cdot z^{20}+(140.045104+161.369678i)\cdot z^{19}+(1274.642064+381.680467i)\cdot z^{18}+(743.821643+5810.837248i)\cdot z^{17}+(16752.870876+10043.831986i)\cdot z^{16}+(41763.432806+29981.366628i)\cdot z^{15}+(23799.211445+106982.985377i)\cdot z^{14}+(189267.174496+34407.932181i)\cdot z^{13}+(152123.766242+236186.810121i)\cdot z^{12}+(198013.830239+280742.352179i)\cdot z^{11}+(342742.413912+83786.997252i)\cdot z^{10}+(37460.901644+301901.877085i)\cdot z^{9}+(195080.181222+100059.575279i)\cdot z^{8}+(94732.085290+90223.058904i)\cdot z^{7}+(27282.343436+57440.498715i)\cdot z^{6}+(24271.198674+3724.963179i)\cdot z^{5}+(682.559499+7237.864131i)\cdot z^{4}+(1528.494152+397.912167i)\cdot z^{3}+(63.349391+237.887291i)\cdot z^{2}+(27.878345+0.897986i)\cdot z^{1}+(0.802320+1.101871i)$



11.7. Polynomial #7

Degree 43 Polynomial: $z^{43}+(0.890-0.19491576i)\cdot z^{42}+(-0.140045104-0.161369678i)\cdot z^{41}+(-0.1274642064-0.381680467i)\cdot z^{40}+(-0.743821643-0.5810837248i)\cdot z^{39}+(-0.16752870876-0.10043831986i)\cdot z^{38}+(0.41763432806+0.29981366628i)\cdot z^{37}+(0.23799211445-0.106982985377i)\cdot z^{36}+(-0.189267174496-0.344079321811i)\cdot z^{35}+(-0.152123766242+0.236186810121i)\cdot z^{34}+(-0.198013830239+0.280742352179i)\cdot z^{33}+(-0.342742413912+0.83786997252i)\cdot z^{32}+(-0.37460901644-0.301901877085i)\cdot z^{31}+(-0.195080181222-0.100059575279i)\cdot z^{30}+(-0.94732085290-0.90223058904i)\cdot z^{29}+(0.27282343436-0.57440498715i)\cdot z^{28}+(0.24271198674-0.3724963179i)\cdot z^{27}+(0.682559499-0.7237864131i)\cdot z^{26}+(-0.1528494152-0.397912167i)\cdot z^{25}+(0.63349391+0.237887291i)\cdot z^{24}+(0.27878345+0.0897986i)\cdot z^{23}+(0.0802320-0.1101871i)\cdot z^{22}+(-0.10+0.0i)\cdot z^{21}+(0.8900000-0.19491576i)\cdot z^{20}+(-0.140045104-0.161369678i)\cdot z^{19}+(-0.1274642064-0.381680467i)\cdot z^{18}+(-0.743821643-0.5810837248i)\cdot z^{17}+(-0.16752870876-0.10043831986i)\cdot z^{16}+(0.41763432806+0.29981366628i)\cdot z^{15}+(0.23799211445-0.106982985377i)\cdot z^{14}+(-0.189267174496-0.344079321811i)\cdot z^{13}+(-0.152123766242+0.236186810121i)\cdot z^{12}+(-0.198013830239+0.280742352179i)\cdot z^{11}+(-0.342742413912+0.83786997252i)\cdot z^{10}+(-0.37460901644-0.301901877085i)\cdot z^{9}+(-0.195080181222-0.100059575279i)\cdot z^{8}+(-0.94732085290-0.90223058904i)\cdot z^{7}+(0.27282343436-0.57440498715i)\cdot z^{6}+(0.24271198674-0.3724963179i)\cdot z^{5}+(0.682559499-0.7237864131i)\cdot z^{4}+(-0.1528494152-0.397912167i)\cdot z^{3}+(0.63349391+0.237887291i)\cdot z^{2}+(0.27878345+0.0897986i)\cdot z^{1}+(0.0802320-0.1101871i)$

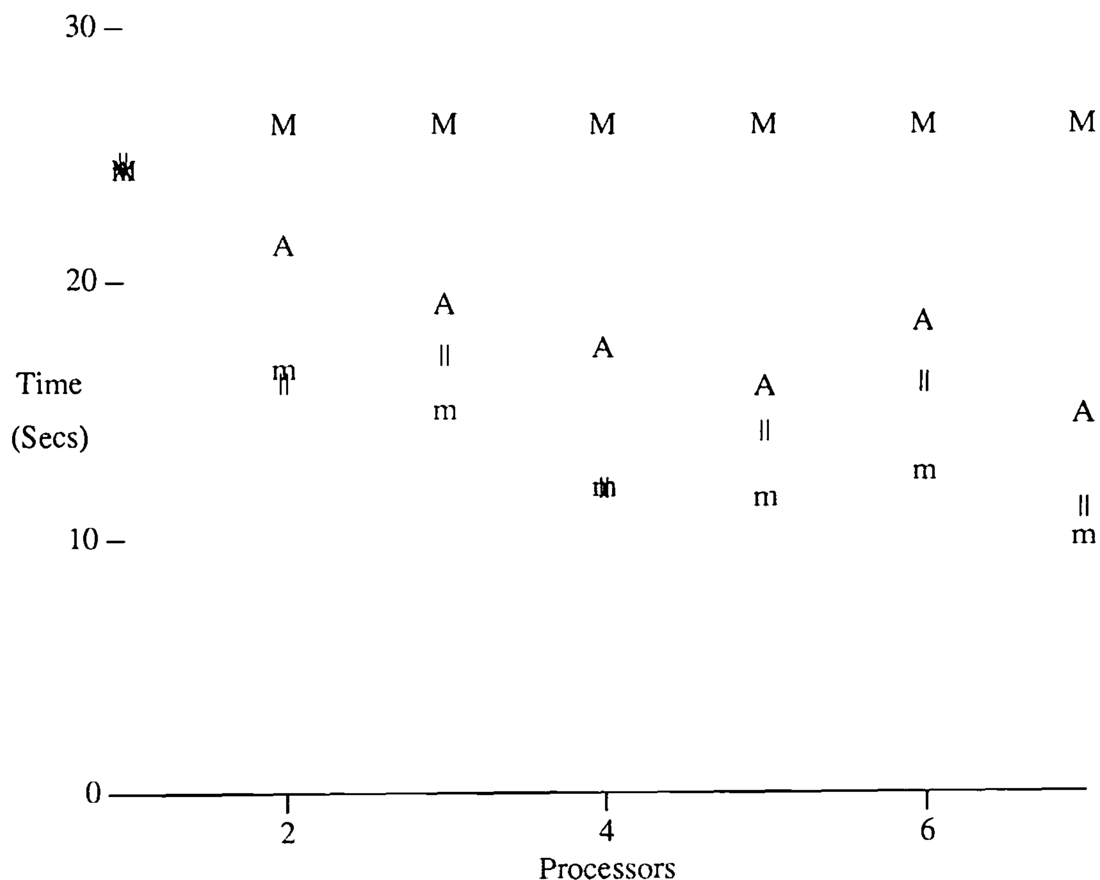


11.8. Polynomial #8

Degree 43 Polynomial:

$z^{43}+$ $(0.890+0.19491576i)\cdot z^{42}+$ $(0.140045104+0.161369678i)\cdot z^{41}+$
$(0.1274642064+0.381680467i)\cdot z^{40}+$ $(0.743821643+0.5810837248i)\cdot z^{39}+$
$(0.16752870876+0.10043831986i)\cdot z^{38}+$ $(0.41763432806+0.29981366628i)\cdot z^{37}+$
$(0.23799211445+0.106982985377i)\cdot z^{36}+$ $(0.189267174496+0.34407932181i)\cdot z^{35}+$
$(0.152123766242+0.236186810121i)\cdot z^{34}+$ $(0.198013830239+0.280742352179i)\cdot z^{33}+$
$(0.342742413912+0.83786997252i)\cdot z^{32}+$ $(0.37460901644+0.301901877085i)\cdot z^{31}+$
$(0.195080181222+0.100059575279i)\cdot z^{30}+$ $(0.94732085290+0.90223058904i)\cdot z^{29}+$
$(0.27282343436+0.57440498715i)\cdot z^{28}+$ $(0.24271198674+0.3724963179i)\cdot z^{27}+$
$(0.682559499+0.7237864131i)\cdot z^{26}+$ $(0.1528494152+0.397912167i)\cdot z^{25}+$

$$(0.63349391+0.237887291i)\cdot z^{24}+ (0.27878345+0.0897986i)\cdot z^{23}+ (0.0802320+0.1101871i)\cdot z^{22}+ (0.10+0.0i)\cdot z^{21}+ (0.890+0.19491576i)\cdot z^{20}+ (0.140045104+0.161369678i)\cdot z^{19}+ (0.1274642064+0.381680467i)\cdot z^{18}+ (0.743821643+0.5810837248i)\cdot z^{17}+ (0.16752870876+0.10043831986i)\cdot z^{16}+ (0.41763432806+0.29981366628i)\cdot z^{15}+ (0.23799211445+0.106982985377i)\cdot z^{14}+ (0.189267174496+0.34407932181i)\cdot z^{13}+ (0.152123766242+0.236186810121i)\cdot z^{12}+ (0.198013830239+0.280742352179i)\cdot z^{11}+ (0.342742413912+0.83786997252i)\cdot z^{10}+ (0.37460901644+0.301901877085i)\cdot z^{9}+ (0.195080181222+0.100059575279i)\cdot z^{8}+ (0.94732085290+0.90223058904i)\cdot z^{7}+ (0.27282343436+0.57440498715i)\cdot z^{6}+ (0.24271198674+0.3724963179i)\cdot z^{5}+ (0.682559499+0.7237864131i)\cdot z^{4}+ (0.1528494152+0.397912167i)\cdot z^{3}+ (0.63349391+0.237887291i)\cdot z^{2}+ (0.27878345+0.0897986i)\cdot z^{1}+ (0.0802320+0.1101871i)$$



12. Appendix IV: Source, Prolog Sorts

```
/*
 * Naive sort
 * Clocksin & Mellish, P. 155
 */
sort(L1,L2) :- permutation(L1,L2), sorted(L2), !.

permutation(L,[H|T]) :-
   append(V,[H|U],L),
   append(V,U,W),
   permutation(W,T).
permutation([],[]).

sorted(L) :- sorted(0,L).

sorted(_,[]).
sorted(N,[H|T]) :- N =< H, sorted(H,T).

/*
 * Insertion sort
 * Clocksin & Mellish, p. 156
 */
insort([],[]).
insort([X|L],M) :- insort(L,N), insortx(X,N,M).

insortx(X,[A|L],[A|M]) :- A =< X, !, insortx(X,L,M).
insortx(X,L,[X|L]).

/*
 * Bubble sort
 * Clocksin & Mellish, p. 156
 */

busort(L,S) :-
   append(X,[A,B|Y],L),
   B < A,
   append(X,[B,A|Y],M),
   busort(M,S).
busort(L,L).

/*
 * Quicksort #1 from
 * Clocksin & Mellish, p.157
 */
split(H,[A|X],[A|Y],Z) :- A =< H, split(H,X,Y,Z).
split(H,[A|X],Y,[A|Z]) :- A > H, split(H,X,Y,Z).
split(_,[],[],[]).

quisort( [],[] ).
quisort( [H|T], S) :-
   split(H,T,A,B),
   quisort(A,A1),
```

```
   quisort(B,B1),
   append(A1,[H|B1], S).


/*
 * Quicksort #2 from
 * Clocksin & Mellish, p.157
 */
quisortx([H|T],S,X) :-
   split(H,T,A,B),
   quisortx(A,S,[H|Y]),
   quisortx(B,Y,X).
quisortx([],X,X).

/*
 * Quicksort
 * from DEC-10 library
 */
qsort([X|L],R0,R) :-
   partition(L,X,L1,L2),
   qsort(L2,R0,R1),
   qsort(L1,[X|R1],R).
qsort([],R,R).

partition([X|L],Y,[X|L1],L2) :- X =< Y, !,
   partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]) :- X > Y, !,
   partition(L,Y,L1,L2).
partition([],_,[],[]).

/* define append(). */
append([], L, L).
append( [H|T], L, [H|V] ) :- append(T,L,V).
```

13. Appendix V: Timings, Prolog Sorts, Large Lists

```
statistics.

qsort( [4,4,8,6,7,9,1,6,6,1,1,9,0,4,3,3,3,4,7,9,4,9,5,9,4,7,0,3,1,1,
7,0,7,5,9,2,1,9,5,1,2,9,5,7,0,7,7,1,0,8,7,3,7,3,3,3,1,7,9,3,1,4,3,3,
1,1,0,9,2,3,5,3,1,4,0,5,7,1,9,4,1,6,3,6,5,2,4,0,9,0,7,5,1,8,1,1,8,2,
8,5,0,4,9,5,5,7,1,8,3,6,1,5,9,0,0,5,7,0,9,9,1,7,9,1,9,1,4,9,7,3,3,7,
3,5,2,7,8,9,1,4,3,6,4,0,2,0,4,6,3,8,4,2,6,0,2,7,2,6,3,7,5,4,8,7,5,2,
7,6,4,3,4,1,4,9,7,5,6,9,2,0,2,0,2,0,0,0,7,9,3,4,6,2,5,5,6,6,6,6,1,5,
1,6,6,5,1,5,2,5,8,2,9,1,9,3,9,3,1,3,4,3,3,9,1,2,2,2,2,6,1,4,6,2,1,2,
4,2,0,1,4,1,5,3,3,6,5,0,3,7,2,8,6,8,3,3,6,6,2,9,2,1,1,8,6,2,9,0,4,2,
3,5,4,7,7,8,9,6,6,3,7,8,3,7,8,5,4,6,7,8,9,3,0,1,2,6,3,8,0,4,1,0,8,2,
1,4,3,7,8,5,4,8,7,3,9,8,8,6,6,4,8,7,9,9,2,3,4,1,1,7,9,9,4,8,5,0,4,3,
3,8,6,6,7,7,8,1,2,5,5,9,4,5,4,1,3,4,7,2,4,6,5,2,4,6,2,3,1,1,9,4,8,8,
1,4,0,1,3,1,6,0,7,1,6,2,8,4,2,7,2,8,1,7,1,3,0,4,2,2,4,7,8,6,9,1,8,5,
6,3,1,2,0,0,1,9,2,3,3,3,6,9,8,9,6,6,9,3,3,5,4,4,3,6,1,6,2,9,3,9,1,3,
1,1,0,4,1,7,3,0,9,5,6,5,0,1,6,9,4,6,6,2,7,5,4,5,5,8,8,7,5,6,4,4,3,4,
5,1,9,1,2,6,9,2,7,9,6,0,0,6,9,3,5,5,1,8,0,9,2,7,1,9,5,6,4,5,0,2,6,4,
2,9,4,0,9,2,8,5,7,4,1,0,8,2,8,3,5,3,5,1,1,8,8,2,7,5], [], X10).

statistics.

quisortx( [4,4,8,6,7,9,1,6,6,1,1,9,0,4,3,3,3,4,7,9,4,9,5,9,4,7,0,3,1,1,
7,0,7,5,9,2,1,9,5,1,2,9,5,7,0,7,7,1,0,8,7,3,7,3,3,3,1,7,9,3,1,4,3,3,
1,1,0,9,2,3,5,3,1,4,0,5,7,1,9,4,1,6,3,6,5,2,4,0,9,0,7,5,1,8,1,1,8,2,
8,5,0,4,9,5,5,7,1,8,3,6,1,5,9,0,0,5,7,0,9,9,1,7,9,1,9,1,4,9,7,3,3,7,
3,5,2,7,8,9,1,4,3,6,4,0,2,0,4,6,3,8,4,2,6,0,2,7,2,6,3,7,5,4,8,7,5,2,
7,6,4,3,4,1,4,9,7,5,6,9,2,0,2,0,2,0,0,0,7,9,3,4,6,2,5,5,6,6,6,6,1,5,
1,6,6,5,1,5,2,5,8,2,9,1,9,3,9,3,1,3,4,3,3,9,1,2,2,2,2,6,1,4,6,2,1,2,
4,2,0,1,4,1,5,3,3,6,5,0,3,7,2,8,6,8,3,3,6,6,2,9,2,1,1,8,6,2,9,0,4,2,
3,5,4,7,7,8,9,6,6,3,7,8,3,7,8,5,4,6,7,8,9,3,0,1,2,6,3,8,0,4,1,0,8,2,
1,4,3,7,8,5,4,8,7,3,9,8,8,6,6,4,8,7,9,9,2,3,4,1,1,7,9,9,4,8,5,0,4,3,
3,8,6,6,7,7,8,1,2,5,5,9,4,5,4,1,3,4,7,2,4,6,5,2,4,6,2,3,1,1,9,4,8,8,
1,4,0,1,3,1,6,0,7,1,6,2,8,4,2,7,2,8,1,7,1,3,0,4,2,2,4,7,8,6,9,1,8,5,
6,3,1,2,0,0,1,9,2,3,3,3,6,9,8,9,6,6,9,3,3,5,4,4,3,6,1,6,2,9,3,9,1,3,
1,1,0,4,1,7,3,0,9,5,6,5,0,1,6,9,4,6,6,2,7,5,4,5,5,8,8,7,5,6,4,4,3,4,
5,1,9,1,2,6,9,2,7,9,6,0,0,6,9,3,5,5,1,8,0,9,2,7,1,9,5,6,4,5,0,2,6,4,
2,9,4,0,9,2,8,5,7,4,1,0,8,2,8,3,5,3,5,1,1,8,8,2,7,5], [], X11).

statistics.

quisort( [4,4,8,6,7,9,1,6,6,1,1,9,0,4,3,3,3,4,7,9,4,9,5,9,4,7,0,3,1,1,
7,0,7,5,9,2,1,9,5,1,2,9,5,7,0,7,7,1,0,8,7,3,7,3,3,3,1,7,9,3,1,4,3,3,
1,1,0,9,2,3,5,3,1,4,0,5,7,1,9,4,1,6,3,6,5,2,4,0,9,0,7,5,1,8,1,1,8,2,
8,5,0,4,9,5,5,7,1,8,3,6,1,5,9,0,0,5,7,0,9,9,1,7,9,1,9,1,4,9,7,3,3,7,
3,5,2,7,8,9,1,4,3,6,4,0,2,0,4,6,3,8,4,2,6,0,2,7,2,6,3,7,5,4,8,7,5,2,
7,6,4,3,4,1,4,9,7,5,6,9,2,0,2,0,2,0,0,0,7,9,3,4,6,2,5,5,6,6,6,6,1,5,
1,6,6,5,1,5,2,5,8,2,9,1,9,3,9,3,1,3,4,3,3,9,1,2,2,2,2,6,1,4,6,2,1,2,
4,2,0,1,4,1,5,3,3,6,5,0,3,7,2,8,6,8,3,3,6,6,2,9,2,1,1,8,6,2,9,0,4,2,
3,5,4,7,7,8,9,6,6,3,7,8,3,7,8,5,4,6,7,8,9,3,0,1,2,6,3,8,0,4,1,0,8,2,
1,4,3,7,8,5,4,8,7,3,9,8,8,6,6,4,8,7,9,9,2,3,4,1,1,7,9,9,4,8,5,0,4,3,
3,8,6,6,7,7,8,1,2,5,5,9,4,5,4,1,3,4,7,2,4,6,5,2,4,6,2,3,1,1,9,4,8,8,
```

```
1,4,0,1,3,1,6,0,7,1,6,2,8,4,2,7,2,8,1,7,1,3,0,4,2,2,4,7,8,6,9,1,8,5,
6,3,1,2,0,0,1,9,2,3,3,3,6,9,8,9,6,6,9,3,3,5,4,4,3,6,1,6,2,9,3,9,1,3,
1,1,0,4,1,7,3,0,9,5,6,5,0,1,6,9,4,6,6,2,7,5,4,5,5,8,8,7,5,6,4,4,3,4,
5,1,9,1,2,6,9,2,7,9,6,0,0,6,9,3,5,5,1,8,0,9,2,7,1,9,5,6,4,5,0,2,6,4,
2,9,4,0,9,2,8,5,7,4,1,0,8,2,8,3,5,3,5,1,1,8,8,2,7,5], X12).

statistics.

insort( [4,4,8,6,7,9,1,6,6,1,1,9,0,4,3,3,3,4,7,9,4,9,5,9,4,7,0,3,1,1,
7,0,7,5,9,2,1,9,5,1,2,9,5,7,0,7,7,1,0,8,7,3,7,3,3,3,1,7,9,3,1,4,3,3,
1,1,0,9,2,3,5,3,1,4,0,5,7,1,9,4,1,6,3,6,5,2,4,0,9,0,7,5,1,8,1,1,8,2,
8,5,0,4,9,5,5,7,1,8,3,6,1,5,9,0,0,5,7,0,9,9,1,7,9,1,9,1,4,9,7,3,3,7,
3,5,2,7,8,9,1,4,3,6,4,0,2,0,4,6,3,8,4,2,6,0,2,7,2,6,3,7,5,4,8,7,5,2,
7,6,4,3,4,1,4,9,7,5,6,9,2,0,2,0,2,0,0,0,7,9,3,4,6,2,5,5,6,6,6,6,1,5,
1,6,6,5,1,5,2,5,8,2,9,1,9,3,9,3,1,3,4,3,3,9,1,2,2,2,2,6,1,4,6,2,1,2,
4,2,0,1,4,1,5,3,3,6,5,0,3,7,2,8,6,8,3,3,6,6,2,9,2,1,1,8,6,2,9,0,4,2,
3,5,4,7,7,8,9,6,6,3,7,8,3,7,8,5,4,6,7,8,9,3,0,1,2,6,3,8,0,4,1,0,8,2,
1,4,3,7,8,5,4,8,7,3,9,8,8,6,6,4,8,7,9,9,2,3,4,1,1,7,9,9,4,8,5,0,4,3,
3,8,6,6,7,7,8,1,2,5,5,9,4,5,4,1,3,4,7,2,4,6,5,2,4,6,2,3,1,1,9,4,8,8,
1,4,0,1,3,1,6,0,7,1,6,2,8,4,2,7,2,8,1,7,1,3,0,4,2,2,4,7,8,6,9,1,8,5,
6,3,1,2,0,0,1,9,2,3,3,3,6,9,8,9,6,6,9,3,3,5,4,4,3,6,1,6,2,9,3,9,1,3,
1,1,0,4,1,7,3,0,9,5,6,5,0,1,6,9,4,6,6,2,7,5,4,5,5,8,8,7,5,6,4,4,3,4,
5,1,9,1,2,6,9,2,7,9,6,0,0,6,9,3,5,5,1,8,0,9,2,7,1,9,5,6,4,5,0,2,6,4,
2,9,4,0,9,2,8,5,7,4,1,0,8,2,8,3,5,3,5,1,1,8,8,2,7,5], X13).

statistics.

busort( [4,4,8,6,7,9,1,6,6,1,1,9,0,4,3,3,3,4,7,9,4,9,5,9,4,7,0,3,1,1,
7,0,7,5,9,2,1,9,5,1,2,9,5,7,0,7,7,1,0,8,7,3,7,3,3,3,1,7,9,3,1,4,3,3,
1,1,0,9,2,3,5,3,1,4,0,5,7,1,9,4,1,6,3,6,5,2,4,0,9,0,7,5,1,8,1,1,8,2,
8,5,0,4,9,5,5,7,1,8,3,6,1,5,9,0,0,5,7,0,9,9,1,7,9,1,9,1,4,9,7,3,3,7,
3,5,2,7,8,9,1,4,3,6,4,0,2,0,4,6,3,8,4,2,6,0,2,7,2,6,3,7,5,4,8,7,5,2,
7,6,4,3,4,1,4,9,7,5,6,9,2,0,2,0,2,0,0,0,7,9,3,4,6,2,5,5,6,6,6,6,1,5,
1,6,6,5,1,5,2,5,8,2,9,1,9,3,9,3,1,3,4,3,3,9,1,2,2,2,2,6,1,4,6,2,1,2,
4,2,0,1,4,1,5,3,3,6,5,0,3,7,2,8,6,8,3,3,6,6,2,9,2,1,1,8,6,2,9,0,4,2,
3,5,4,7,7,8,9,6,6,3,7,8,3,7,8,5,4,6,7,8,9,3,0,1,2,6,3,8,0,4,1,0,8,2,
1,4,3,7,8,5,4,8,7,3,9,8,8,6,6,4,8,7,9,9,2,3,4,1,1,7,9,9,4,8,5,0,4,3,
3,8,6,6,7,7,8,1,2,5,5,9,4,5,4,1,3,4,7,2,4,6,5,2,4,6,2,3,1,1,9,4,8,8,
1,4,0,1,3,1,6,0,7,1,6,2,8,4,2,7,2,8,1,7,1,3,0,4,2,2,4,7,8,6,9,1,8,5,
6,3,1,2,0,0,1,9,2,3,3,3,6,9,8,9,6,6,9,3,3,5,4,4,3,6,1,6,2,9,3,9,1,3,
1,1,0,4,1,7,3,0,9,5,6,5,0,1,6,9,4,6,6,2,7,5,4,5,5,8,8,7,5,6,4,4,3,4,
5,1,9,1,2,6,9,2,7,9,6,0,0,6,9,3,5,5,1,8,0,9,2,7,1,9,5,6,4,5,0,2,6,4,
2,9,4,0,9,2,8,5,7,4,1,0,8,2,8,3,5,3,5,1,1,8,8,2,7,5], X14).

statistics.

/*
 * can't be done; would take forever!!!
sort( [4,4,8,6,7,9,1,6,6,1,1,9,0,4,3,3,3,4,7,9,4,9,5,9,4,7,0,3,1,1,
7,0,7,5,9,2,1,9,5,1,2,9,5,7,0,7,7,1,0,8,7,3,7,3,3,3,1,7,9,3,1,4,3,3,
1,1,0,9,2,3,5,3,1,4,0,5,7,1,9,4,1,6,3,6,5,2,4,0,9,0,7,5,1,8,1,1,8,2,
8,5,0,4,9,5,5,7,1,8,3,6,1,5,9,0,0,5,7,0,9,9,1,7,9,1,9,1,4,9,7,3,3,7,
3,5,2,7,8,9,1,4,3,6,4,0,2,0,4,6,3,8,4,2,6,0,2,7,2,6,3,7,5,4,8,7,5,2,
7,6,4,3,4,1,4,9,7,5,6,9,2,0,2,0,2,0,0,0,7,9,3,4,6,2,5,5,6,6,6,6,1,5,
```

```
1,6,6,5,1,5,2,5,8,2,9,1,9,3,9,3,1,3,4,3,3,9,1,2,2,2,2,6,1,4,6,2,1,2,
4,2,0,1,4,1,5,3,3,6,5,0,3,7,2,8,6,8,3,3,6,6,2,9,2,1,1,8,6,2,9,0,4,2,
3,5,4,7,7,8,9,6,6,3,7,8,3,7,8,5,4,6,7,8,9,3,0,1,2,6,3,8,0,4,1,0,8,2,
1,4,3,7,8,5,4,8,7,3,9,8,8,6,6,4,8,7,9,9,2,3,4,1,1,7,9,9,4,8,5,0,4,3,
3,8,6,6,7,7,8,1,2,5,5,9,4,5,4,1,3,4,7,2,4,6,5,2,4,6,2,3,1,1,9,4,8,8,
1,4,0,1,3,1,6,0,7,1,6,2,8,4,2,7,2,8,1,7,1,3,0,4,2,2,4,7,8,6,9,1,8,5,
6,3,1,2,0,0,1,9,2,3,3,3,6,9,8,9,6,6,9,3,3,5,4,4,3,6,1,6,2,9,3,9,1,3,
1,1,0,4,1,7,3,0,9,5,6,5,0,1,6,9,4,6,6,2,7,5,4,5,5,8,8,7,5,6,4,4,3,4,
5,1,9,1,2,6,9,2,7,9,6,0,0,6,9,3,5,5,1,8,0,9,2,7,1,9,5,6,4,5,0,2,6,4,
2,9,4,0,9,2,8,5,7,4,1,0,8,2,8,3,5,3,5,1,1,8,8,2,7,5], X15).
 *
 * statistics.
 */
```

14. Appendix VI: Sort Performance, Small Lists

```
statistics.

qsort( [4,4,8,6,7,9,1,6,6,1,1,9,0], [], X30).

statistics.

quisortx( [4,4,8,6,7,9,1,6,6,1,1,9,0], X31, []).

statistics.

quisort( [4,4,8,6,7,9,1,6,6,1,1,9,0], X32).

statistics.

insort( [4,4,8,6,7,9,1,6,6,1,1,9,0], X33).

statistics.

busort( [4,4,8,6,7,9,1,6,6,1,1,9,0], X34).

statistics.

sort( [4,4,8,6,7,9,1,6,6,1,1,9,0], X35).

statistics.
```

15. Appendix VII: Naive Sort Performance

```
statistics.

sort( [4], X30).

statistics.

sort( [4,4], X31).

statistics.

sort( [4,4,8], X32).

statistics.

sort( [4,4,8,6], X33).

statistics.

sort( [4,4,8,6,7], X34).

statistics.

sort( [4,4,8,6,7,9], X35).

statistics.

sort( [4,4,8,6,7,9,1], X36).

statistics.

sort( [4,4,8,6,7,9,1,6], X37).

statistics.

sort( [4,4,8,6,7,9,1,6,6], X38).

statistics.

sort( [4,4,8,6,7,9,1,6,6,1], X39).

statistics.
```

16. Appendix VIII: Performance on Small, Ordered Lists

```
statistics.

qsort( [0,1,2,3,4,5,6,7,8,9,10,11,12], [], X30).

statistics.

quisortx( [0,1,2,3,4,5,6,7,8,9,10,11,12], X31, []).

statistics.

quisort( [0,1,2,3,4,5,6,7,8,9,10,11,12], X32).

statistics.

insort( [0,1,2,3,4,5,6,7,8,9,10,11,12], X33).

statistics.

busort( [0,1,2,3,4,5,6,7,8,9,10,11,12], X34).

statistics.

sort( [0,1,2,3,4,5,6,7,8,9,10,11,12], X35).

statistics.
```

17. Appendix IX: Program to estimate memory speeds

```
/*
 * program to estimate memory speeds.
 * One argument: size of memory to {mem|b}copy, once.
 * Strategy:
 * We page it in via the first loop.
 * The loop is set up to page, and not utilize the CPU's data cache.
 * Then, the time is obtained, we do the copy (in place) and,
 * the resulting time is calculated.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/times.h>
#include <errno.h>
#include <memory.h>

#ifndef NBPC
#define PAGE_SIZE 2048
#else
#define PAGE_SIZE NBPC
#endif

#ifndef NULL
#define NULL 0
#endif

main( argc, argv )
int argc;
char *argv[];
{
   int size;
   char *p1, *p2, *malloc();
   struct tms tb1, tb2;
   long clock, times();

   if( argc <= 1 )
      error( "usage: mem_speed size\n" );

   size = atoi( argv[1] );
   if( (p1 = malloc( size ) ) == (char *) NULL )
      error( "mem_speed: can't allocate memory.\n" );

   for( p2 = p1; p2 < &p1[size]; p2 = &p2[PAGE_SIZE] )
      *p2 = ' ';

   p2 = p1;

   clock = times( &tb1 );
```

```
   memcpy( p2, p1, size );

   clock = times( &tb2 ) - clock;

   printf( "Real: %.2f, User: %.2f, System: %.2f\n",
    (1.0* (double)clock)/(1.0*(double)HZ),
    (1.0*(tb2.tms_utime-tb1.tms_utime))/(1.0*HZ),
    (1.0*(tb2.tms_stime-tb1.tms_stime))/(1.0*HZ) );

   exit(0);
}

error( string )
char *string;
{
   fprintf( stderr, "%s", string );
   exit( 1 );
}
```

18. Appendix X: Lower bound affects dispersion

```
#include <stdio.h>
#include <math.h>
#include <sys/types.h>
#include <sys/times.h>
#include <sys/time.h>

#define MILLION 1000000
#define TRUE 1
#define FALSE 0
#define MAX_SIZE 100000
#define CCOST 5     /* cost of comparison */

int S[MAX_SIZE],
     Ticks[MAX_SIZE],
     P[MAX_SIZE],
     Copy[MAX_SIZE],
     SSize = -1, PSize = -1;
int Perm = FALSE;
int Least = FALSE;

/*
 * Program to test performance of various quicksorting variations,
 * but mostly the application of randomness to partition element
 * selection.
 *
 * Options:
 * -p: generate a permutation to sort rather than a random (might
 *     contain duplicates) list.
 * -s: specifies size of permutation or list to generate. Limited
 *     to value of symbol MAX_SIZE.
 * -c: count of attempts to sort the list.
 * -l: use element of list with smallest index as the pivot rather
 *     than selecting partition at random: see mysort(), below.
 *
 */

main( argc, argv )
int argc;
char *argv[];
{
   double total, scaler, dSize, floor(), drand48();
   register int i, j, k;
   int min, max;
   long clock;
   struct timeval tv1;
   struct timezone tz;

   while( argc > 1 )
   {
      if( argv[argc-1][0] != '-' )
         usage();
```

```
      switch( argv[argc-1][1] )
      {
      case 's':
         PSize = atoi( &argv[argc-1][2] );
         if( PSize < 1 || PSize > MAX_SIZE )
            usage();
         break;

      case 'c':
         SSize = atoi( &argv[argc-1][2] );
         if( SSize < 1 || SSize > MAX_SIZE )
            usage();
         break;

      case 'p': /* use random permutation rather than
                 * random list
                 */
         Perm = TRUE;
         break;

      case 'l': /* use element of smallest index in sort
             * rather than a random element
             */
         Least = TRUE;
         break;

      default:
         usage();
      }
      argc -= 1;
   }

   if( PSize == -1 )
      PSize = MAX_SIZE;
   if( SSize == -1 )
      SSize = MAX_SIZE;

   dSize = (double) PSize;

   gettimeofday( &tvl, &tz );
   clock = MILLION*tvl.tv_sec+tvl.tv_usec;
   srand48( clock );

   if( Perm == TRUE )
   {
      for( i = 0; i < PSize; i += 1 )
         P[i] = i;

      for( i = 0; i < PSize; i += 1 )
      {
         k = (int) floor( drand48()*dSize );
         j = P[i]; P[i] = P[k]; P[k] = j;
      }
   }
```

```
    else
    {
       for( i = 0; i < PSize; i += 1 )
          P[i] = (int) floor( drand48()*dSize );
    }

    for( i = 0; i < SSize; i += 1 )
    {
       for( j = 0; j < PSize; j += 1 )
          Copy[j] = P[j];

       gettimeofday( &tv1, &tz );
       clock = MILLION*tv1.tv_sec+tv1.tv_usec;
       Ticks[i] = clock;

       mysort( Copy, 0, PSize-1 );

       gettimeofday( &tv1, &tz );
       clock = MILLION*tv1.tv_sec+tv1.tv_usec;
       Ticks[i] = clock - Ticks[i];
    }

    /* dump statistics */
    scaler = 1.0/((double) MILLION);
    min = max = Ticks[0];
    /* we pre-scale Ticks to prevent overflow */
    total = scaler*Ticks[0];

    for( i = 1; i < SSize; i += 1 )
    {
       if( Ticks[i] < min )
          min = Ticks[i];
       if( Ticks[i] > max )
          max = Ticks[i];
       total += scaler*Ticks[i];
    }

    printf( "min: %g sec, max: %g sec, avg: %g sec.\n",
       scaler* (double) min,
       scaler* (double) max,
       total / (double) SSize );

    exit( 0 );
}

usage()
{
    fprintf( stderr, "Usage: rand_qsort [-ssize] [-ccount] [-l] [-p]\n" );
    fprintf( stderr, "size & count <= %d\n", MAX_SIZE );
    exit( 1 );
}

/* implementation of quicksort which randomizes.
 * There are some special tricks applied here which makes the
```

```
 * code more complex (and faster) than many published versions.
 * In particular, if there are more than one occurrence of the
 * partitioning element, they create a "center" to the array
 * which must be removed in order for the partitioning to be correct.
 * "Software Tools" quicksort avoids this problem by always
 * choosing the last element as the partition, and pushing
 * other elements up against it.
 *
 * options: Least == TRUE; choose least element rather than
 * a random element.
 */

mysort( array, start, finish )
int array[], start, finish;
{
   int i, random, pivot, hi, lo, tmp, exchanges;
   double drand48(), floor();

   if( Least == TRUE )
      random = start;
   else
      random = start + (int)floor( drand48()*( (double)(finish-start+1) ) );

   pivot = array[random];
   lo = start;   /* upper bnd, current set of values < pivot */
   hi = finish;    /* lower bnd, current set of values < pivot */

   while( lo < hi )
   {
         while( (lo < hi) && (compare( array[lo], pivot ) < 0 ) )
          lo = lo+1 ;

      while( (lo < hi) && ( compare( array[hi], pivot ) > 0 ) )
         hi = hi-1 ;

      if( lo < hi )
      {

         /* this block of code insures that all of
          * the elements with value == pivot are
          * bunched and ordered correctly.
          */

         if( (compare( array[hi], pivot ) == 0 ) &&
             ( compare( array[lo], pivot ) == 0 ))
         {
            exchanges = 0;
            for( i = lo+1; i <= hi-1; i += 1 )
            {
               if( compare( array[i], pivot ) < 0 )
               {
                  tmp = array[lo];
                  array[lo] = array[i];
                  array[i] = tmp;
```

```
                    lo += 1 ;
                    ++exchanges;
                 }
                 else if( compare( array[i], pivot ) > 0 )
                 {
                    tmp = array[hi];
                    array[hi] = array[i];
                    array[i] = tmp;
                    hi -= 1 ;
                    ++exchanges;
                 }
              }
              if( exchanges == 0 )
                 goto recurse;
           }
           else
           {
              tmp = array[lo];
              array[lo] = array[hi];
              array[hi] = tmp;
           }
        }
     }

recurse:
   if( lo-start > 1 )
      mysort( array, start, lo-1  );
   if( finish-hi > 1 )
      mysort( array, hi+1, finish );

   return;
}

/*
 * this makes comparison costs significant compared to
 * random # generation costs.
 * CCOST can be altered to change significance of
 * comparison costs.
 */

int
compare( i, j )
int i, j;
{
   int waste, diff;
   double drand48(), floor();

   for( waste = 0; waste <= CCOST; waste += 1 )
        diff = (int)floor( drand48()*( (double)(waste-j) ) );

   diff = i - j;

   if( diff < 0 )
      return( -1 );
```

```
   if( diff > 0 )
      return( 1 );
   return( 0 );
}

$ for i in 1 5 10 50 100 500
> do
>  rand_qsort -s500 -c${i}
> done
min: 4.88386 sec, max: 4.88386 sec, avg: 4.88386 sec.
min: 4.60353 sec, max: 5.2892 sec, avg: 4.78954 sec.
min: 4.5822 sec, max: 5.63769 sec, avg: 4.96763 sec.
min: 4.68141 sec, max: 6.8771 sec, avg: 5.49659 sec.
min: 4.58186 sec, max: 12.6386 sec, avg: 5.52683 sec.
min: 4.45218 sec, max: 14.1808 sec, avg: 5.46629 sec.
```

19. Appendix XI: do_elim Script

```
if [ ! -f do_elim ]
then
   echo "Making do_elim."
   make do_elim
fi
if [ ! -f do_elim ]
then
   echo "No do_elim. Exiting."
   exit 1
fi
echo "size do_elim:"
size do_elim
if [ -f /tmp/niceit ]
then
   rm -f /tmp/niceit
   exec nice -20 script
fi

for Groups in 0 1
do
   for Files in 0 5 10 15
   do
      for Asynch in 0 1
      do
         for Work in 0 1
         do
            for Size in 0 1000 3162 10000 31622 100000
            do
               for Dirty in 0 1
               do
                  for Procs in 1 2 4 8 16
                  do
                     Output=`do_elim\
                         -g${Groups}\
                         -f${Files}\
                         -s${Size}\
                         -w${Work}\
                         -a${Asynch}\
                         -d${Dirty}\
                         -p${Procs}`
                     echo\
                        "do_elim\
                        -g${Groups}\
                        -f${Files}\
                        -s${Size}\
                        -w${Work}\
                        -a${Asynch}\
                        -d${Dirty}\
                        -p${Procs}:\
                        ${Output}" |\
                     sed -e 's/  */ /g'
```

```
                        done
                    done
                done
            done
        done
    done
done
```

20. Appendix XII: do_elim.c

```
#include <errno.h>
#include <sys/param.h>
#include <sys/types.h>
#include <sys/times.h>
#include <sys/signal.h>
#include <sys/stat.h>

#ifndef NPROC
#define NPROC 100
#endif

#ifndef NBPC
#define PAGE_SIZE 2048
#else
#define PAGE_SIZE NBPC
#endif

#ifndef NULL
#define NULL 0
#endif

#define EOS '\0'
#define EVER ;;
#define HUGE 0x10000000
#define TRUE 1
#define FALSE 0
#define TMP_PREFIX "/tmp/d_eFXXXXXX"
#define REP_COUNT 100

int
   ProcIdTable[NPROC],
   Groups = 0,
   Files = 0,
   Size = 0,
   Work = 0,
   Asynch = 0,
   Dirty = 0,
   Procs = 0;

char *TmpFileNames[NOFILE];

/*
 * do_elim:
 *  To evaluate costs of sibling elimination.
 *  Eliminates siblings with a SIG_INTR signal (no dump)
 *
 * flags:
 *  -g[01]: 1-use process group feature of kill(). Default: 0
 *  -p[n]: # of processes to spawn. Default: 0
 *  -s[n]: bytes of memory to allocate. Default: 0
 *  -f[n]: open files per process. Default: 0
```

```
 *   -w[01]: work mix: sleep vs. sleep + busy idle loop. Default: 0
 *   -a[01]: asynchronous: (kind of hard to measure!). Default: 0
 *   -d[01]: dirty all pages (to defeat c-o-w management). Default: 0
 */

main( argc, argv )
int argc;
char *argv[];
{
   struct tms tb1, tb2;
   long clock, times(), real_t, user_t, sys_t;
   double scaler;
   int i;
   char *osbrk, *sbrk();

   for( ++argv, --argc; argc > 0; ++argv, --argc )
   {
      if( argv[0][0] != '-' )
         usage();
      switch( argv[0][1] )
      {
      case 'g':
         Groups = getnum( &argv[0][2], 0, 1 );
         break;

      case 'p':
         Procs = getnum( &argv[0][2], 0, NPROC );
         break;

      case 's':
         Size = getnum( &argv[0][2], 0, HUGE);
         break;

      case 'f':
         Files = getnum( &argv[0][2], 0, NOFILE );
         break;

      case 'w':
         Work = getnum( &argv[0][2], 0, 1 );
         break;

      case 'a':
         Asynch = getnum( &argv[0][2], 0, 1 );
         break;

      case 'd':
         Dirty = getnum( &argv[0][2], 0, 1 );
         break;

      default:
         usage();
         break;
      }
   }
```

```
   /* set up new process group so that our parent not killed */
   if( setpgrp() < 0 )
        fail( "Can't set new process group.\n" );

   /* timing phase; do repetitions help? */

   for( i = 0, real_t = user_t = sys_t = 0.0;
      i < REP_COUNT;
      i += 1 )
   {
      startup();

      clock = times( &tb1 );

      eliminate();

      clock = times( &tb2 ) - clock;

      real_t += clock;
      user_t += (tb2.tms_utime-tb1.tms_utime);
      sys_t += (tb2.tms_stime-tb1.tms_stime);

      cleanup();
   }

   scaler = 1.0/(((double) HZ));

   printf( "Real: %.3f, User: %.3f, System: %.3f\n",
      ((double) real_t)*scaler,
      ((double) user_t)*scaler,
      ((double) sys_t)*scaler );

   exit(0);
}

int
getnum( str, min, max )
char *str;
int min, max;
{
   int val;

   val = convt( str );
   if( val < min || val > max )
      usage();
   return( val );
}

usage()
{
   fail( "Usage: do_elim [-g] [-p] [-s] [-f] [-w] [-d] [-a]\n" );
}

int
```

```
convt( str )
char *str;
{
   int i;

   for( i = 0; *str != EOS; ++str )
   {
      if( *str >= '0' && *str <= '9' )
         i = 10*i + (*str - '0' );
      else
         return( -1 );   /* invalid char. */
   }
   return( i );
}

startup()
{
   char *ptr, *emalloc(), *temp_file();
   int i, pid;

   ptr = emalloc( Size );

   for( i = 0; i < Files; i += 1 )
   {
        TmpFileNames[i] = temp_file();
      if( creat( TmpFileNames[i], 0 ) < 0 )
      {
         perror( TmpFileNames[i] );
         fail( "Can't create temporary file.\n" );
      }
   }

   /* set up signals so message not ignored */
   signal( SIGTERM, SIG_DFL );

   for( i = 0; i < Procs; i += 1 )
   {
      switch( (pid = fork()) )
      {
      case -1:
         fail( "Can't fork.\n" );
         break;

      case 0:      /* in child. Dirty and Work if necessary */
         if( Dirty )
            write_it( ptr, Size );

         for( EVER )
         {
            if( Work )
               iterate( 10000 );
            sleep( 1 );
         }
         break;
```

```
      default:
         ProcIdTable[i] = pid;
         break;
      }
   }

   /* only the Parent should ever be able to get here */
   free( ptr );
   return;
}

iterate( count )
register int count;
{
   while( count-- )
      ;
   return;
}

eliminate()
{
   int i, status;

   if( Groups )
   {
      /* set up signals so *we* don't get eliminated */
      signal( SIGTERM, SIG_IGN );
      if( kill( 0 , SIGTERM ) < 0 )
         fail( "Group kill failed.\n" );
   }
   else
   {
      for( i = 0; i < Procs; i += 1 )
      {
         if( kill( ProcIdTable[i], SIGTERM ) < 0 )
            fail( "Kill of proc failed.\n" );
      }
   }

   if( Asynch )
      return;
   else
   {
      for( i = Procs; i < 0; i = i-1 )
      {
         if( wait( &status ) < 0 )
         {
            if( errno == ECHILD )
                continue;
            else
               fail( "Wait failed, !ECHILD\n" );
         }

         if( (0xFFFF & status) != SIGTERM )
```

```
            {
               printf( "status: 0x%x\n", status );
               fail( "Process terminated, wrong reason.\n" );
            }
         }
      }
      return;
}

int
cleanup()
{
   int i;

   for( i = 0; i < Files; i += 1 )
   {
      unlink( TmpFileNames[i] );
      free( TmpFileNames[i] );
      close( i+3 ); /* ??? */
   }

   while( wait( &i) >= 0 )
      ;

   if( errno != ECHILD )
      fail( "Cleanup: Wait failed, != ECHILD\n" );

   return;
}

char *
strsave( str )
char *str;
{
   char *p, *emalloc();

   p = emalloc( strlen( str ) + 1 );
   strcpy( p, str );
   return( p );
}

char *
temp_file()
{
   char *s, *strsave();
   void unique_temp();

   s = strsave( TMP_PREFIX );
   unique_temp( s );
   return( s );
}

void
unique_temp( s )
```

```
char *s;
{
   static long random;
   long work;
   char *p;
   register i;

   p = &s[strlen( s )-6];

   do
   {
      random = (random + 32647 + getpid() ) * 32653;
      work = random;

      for( i=0; i<6; i++ )
      {
         p[i] = "abcdefghijklmnopqrstuvwxyz012345"[ work & 0x1F ];
         work = work >> 5;
      }

   } while( exists( s ) == TRUE );

   return;
}

int
exists( name )
char *name;
{
   struct stat sb;
   int ret;
   extern errno;

   ret = stat( name, &sb );
   if( ret < 0 )
      if( errno == ENOENT )
         return( FALSE );

   return( TRUE );
}

char *
emalloc( size )
unsigned int size;
{
   char *ptr, *malloc();

   if( (ptr = malloc( size)) == (char *) NULL )
   {
      fail( "No Memory. bailing out!\n" );
   }

   return( ptr );
}
```

```
write_it( mem, size )
char *mem;
int size;
{
   double write_count = 0.0, write_size = 0.0;

   write_size = 1.0 * (double) size;

   while( write_count < write_size )
   {
      *mem = EOS;
      mem = &mem[PAGE_SIZE];
      write_count += (double) PAGE_SIZE;
   }
   return;
}

fail( string )
char *string;
{
   write( 2, string, strlen( string ) );
   exit( 1 );
}
```

21. Appendix XIII: "C" version of Jenkins-Traub algorithm

```
/*
 *
 * This is the Jenkins-Traub root-finding
 * algorithm from CACM, February 1972, V 15, Number 2,
 * pages 97-99. It is listed as ''Algorithm 419:
 * Zeros of a Complex Polynomial''
 *
 * The algorithm has been recoded in "C",
 * which necessitated some changes.
 * To activate the debugging statements,
 * compile with -DEBUG.
 *
 * the machine constants have been changed to
 * reflect the IEEE floating point standard,
 * rather than the IBM/360.
 * a minor bug in cmod() was repaired; it had
 * divided by zero if given the complex origin
 * as an argument.
 * added angle (in radians) as an argument to cpoly().
 * all changes noted in ACM corrigendum (CACM 3/74)
 * have been applied.
 * the number of fixed and variable shifts has been
 * increased so that the algorithm is more effective
 * at multiple roots.
 *
 * if you find errors, contact me, Jonathan M. Smith,
 * at 450 Computer Science, Columbia University, NY, NY 10027
 * or jms@close.cs.columbia.edu, on the ARPAnet
 *
 * remainder is straight out of listing:
 * finds the zeros of a complex polynomial.
 * opr, opi - double precision vectors of real and
 * imaginary parts of the coefficients in
 * order of decreasing powers.
 * zeror, zeroi - output double precision vectors of
 * real and imaginary parts of the zeros.
 * fail - output logical parameter, true only if
 * leading coefficient is zero or if cpoly
 * has found fewer than degree zeros.
 * the program has been written to reduce the chance of overflow
 * occurring. if it does occur, there is still a possibility that
 * the zerofinder will work provided the overflowed quantity is
 * replaced by a large number.
 */

/* includes */
#include <math.h>
#include <stdio.h>

/* definitions */
#define MAX_DEGREE 50    /* biggest polynomial it can solve */
```

```
#define TRUE 1
#define FALSE 0
#define dabs(_x) ((_x<0.0)?(-_x):(_x))
#ifndef M_SQRT2
#define M_SQRT2     1.41421356237309504880
#define M_SQRT1_2   0.70710678118654752440
#endif

/* Globals */

double smalno, sr, si, tr, ti, pvr, pvi, are, mre, eta, infin,
base, errev(), cmod(), scale(), cauchy(), sqrt(), cos(), sin(),
pr[MAX_DEGREE],
pi[MAX_DEGREE],
hr[MAX_DEGREE],
hi[MAX_DEGREE],
qpr[MAX_DEGREE],
qpi[MAX_DEGREE],
qhr[MAX_DEGREE],
qhi[MAX_DEGREE],
shr[MAX_DEGREE],
shi[MAX_DEGREE];

int nn;

cpoly(opr,opi,degree,zeror,zeroi,fail,angle)
double    *opr, *opi, *zeror, *zeroi, angle;
int *fail, degree;
{
     double xx,yy,cosr,sinr,xxx,zr,zi,bnd;
     int i, conv, cnt1, cnt2, idnn2;

     mcon();
     are = eta;
     mre = 2.0*M_SQRT2*eta;
#ifdef EBUG
     printf( "mre=%e, are=%e\n", mre, are );
#endif
     xx = M_SQRT1_2; /* 0.70710678 */
     yy = -xx;
     cosr = cos( angle ); /* -.069756474 */
     sinr = sin( angle ); /* .99756405 */
     *fail = FALSE;
     nn = degree+1;

     if( opr[0] != 0.0 || opi[0] != 0.0 ) /* fail if lead coeff. zero */
     {
          /* remove any zeros at origin */
          while( !( opr[nn-1] != 0.0 || opi[nn-1] != 0.0 ) )
          {
               idnn2 = degree-nn+2;
               zeror[idnn2-1] = 0.0;
               zeroi[idnn2-1] = 0.0;
               nn = nn - 1;
```

```
        }
    }
    else
    {
        *fail = TRUE;
        return;
    }

    /* copy coefficients */
    for( i = 1; i <= nn; i += 1 )
    {
        pr[i-1] = opr[i-1];
        pi[i-1] = opi[i-1];
        shr[i-1] = cmod(pr[i-1], pi[i-1]);
    }

    /* scale the polynomial */
    bnd = scale( shr );
    if( bnd != 1.0 )
    {
        for( i=1; i <= nn; i += 1 )
        {
            pr[i-1] = bnd*pr[i-1];
            pi[i-1] = bnd*pi[i-1];
        }
    }

    /* start the algorithm */
    while( nn > 2 )
    {
        for( i=1; i <= nn; i += 1 )
        {
            shr[i-1] = cmod(pr[i-1], pi[i-1]);
        }
        bnd = cauchy( shr, shi);

        /* 2 major passes, different sequences of shifts */
        for( cnt1 = 1; cnt1 <= 2; cnt1 += 1 )
        {
            noshft( 5 ); /* first stage, no shift */

            /* inner loop to select a shift */
            for( cnt2 = 1; cnt2 <= 9; cnt2 += 1 )
            {

                /* shift is chosen with modulus bnd
                 * and amplitude rotated by angle
                 * degrees from the previous shift
                 */
                xxx = cosr*xx-sinr*yy;
                yy = sinr*xx+cosr*yy;
                xx = xxx;
                sr = bnd*xx;
                si = bnd*yy;
```

```
#ifdef EBUG
                    printf( "shift: %g+ %gi\n",
                         sr,si );
#endif

                    /* 2nd stage, fixed shift */
                    fxshft(10*cnt2, &zr, &zi, &conv);
                    if( conv )
                    {
                         idnn2 = degree-nn+2;
                         zeror[idnn2-1] = zr;
                         zeroi[idnn2-1] = zi;
#ifdef EBUG
                         printf( "zero: %g+ %gi\n", zr,zi );
#endif
                         nn = nn-1;
                         for( i = 1; i <=nn; i += 1 )
                         {
                              pr[i-1] = qpr[i-1];
                              pi[i-1] = qpi[i-1];
                         }
                         goto found_root; /* necessary evil */
                    }
               }
          }
          *fail = TRUE;
          return;
          found_root:
               ;
     }

     if( nn >= 1 )
          cdivid( -pr[1], -pi[1], pr[0], pi[0],
          &zeror[degree-1], &zeroi[degree-1] );
#ifdef EBUG
     printf( "nn was %d at return.\n",nn );
#endif
     return;
}

/*
 * computes the derivative polynomial as the initial h
 * polynomial and computes l1 no-shift h polynomials.
 */
noshft(l1)
int l1;
{
     double xni,t1, t2, dn;
     int i, n, nm1, jj, j;

     n = nn - 1;
     nm1 = n - 1;
     dn = (double) n;
```

```
    for( i = 1; i <=n; i += 1 )
    {
         xni = (double) (nn - i);
         hr[i-1] = xni*pr[i-1]/dn;
         hi[i-1] = xni*pi[i-1]/dn;
    }
    for( jj = 1; jj <= l1; jj += 1 )
    {
         if(cmod(hr[n-1],hi[n-1])> eta*10.0*cmod(pr[n-1],pi[n-1]))
         {
              cdivid(-pr[nn-1],-pi[nn-1],hr[n-1],hi[n-1],&tr,&ti);
              for( i = 1; i <= nm1; i += 1 )
              {
                   j = nn-i;
                   t1 = hr[j-2];
                   t2 = hi[j-2];
                   hr[j-1] = tr*t1-ti*t2+pr[j-1];
                   hi[j-1] = tr*t2+ti*t1+pi[j-1];
              }
              hr[0] = pr[0];
              hi[0] = pi[0];
         }
         else
         {
              for( i=1; i <= nm1; i += 1 )
              {
                   j = nn-i;
                   hr[j-1] = hr[j-2];
                   hi[j-1] = hi[j-2];
              }
              hr[0] = 0.0;
              hi[0] = 0.0;
         }
    }
    return;
}

/*
 * computes l2 fixed-shift h polynomials and tests for
 * convergence
 * initiates a variable-shift iteration and returns with the
 * approximate zero if successful.
 * l2 - limit of fixed shift steps
 * zr,zi - approx zero if conv is .true.
 * conv logical indicating convergence of stage 3 iteration
 */

fxshft(l2,zr,zi,conv)
int l2, *conv;
double *zr, *zi;
{
    double otr,oti,svsr,svsi;
    int test,pasd,bool;
    int n, i, j;
```

```
	n = nn-1;
	polyev(nn,sr,si,pr,pi,qpr,qpi,&pvr,&pvi);  /* evaluate p at s */
	test = TRUE;
	pasd = FALSE;
	calct( &bool ); /* 1st t = -p(s)/h(s) */

	/* main loop, 2nd stage step */
	for( j = 1; j <=12; j += 1 )
	{
		otr = tr;
		oti = ti;
		nexth( &bool ); /* next polynomial */
		calct( &bool ); /* new t */
		*zr = sr+tr;
		*zi = si+ti;

		/* convergence test */
		if( bool || (!test) || j == 12)
			continue;

		if( cmod(tr-otr,ti-oti) >= .5*cmod(*zr,*zi)) /* weak conv. */
		{
			pasd = FALSE;
			continue;
		}
		if( ! pasd )
		{
			pasd = TRUE;
			continue;
		}
		for( i = 1; i <=n; i += 1 )
		{
			shr[i-1] = hr[i-1];
			shi[i-1] = hi[i-1];
		}
		svsr = sr;
		svsi = si;
		vrshft(60,zr,zi,conv);
		if( *conv)
			return;
		test = FALSE;
		for( i = 1; i <= n; i += 1 )
		{
			hr[i-1] = shr[i-1];
			hi[i-1] = shi[i-1];
		}
		sr = svsr;
		si = svsi;
		polyev(nn,sr,si,pr,pi,qpr,qpi,&pvr,&pvi);
		calct( &bool );
	}
	vrshft(50, zr, zi, conv);
	return;
}
```

```
/*
 * carries out the third stage iteration
 * l3 - limit of steps in stage 3.
 * zr,zi - on entry contains the initial iterate, if the
 * iteration converges it contains the final iterate
 * on exit.
 * conv - .true. if iteration converges
 */

vrshft(l3,zr,zi,conv)
int l3, *conv;
double *zr, *zi;
{
     double mp,ms,omp,relstp,r1,r2,tp, twenerr;
     int b,bool;
     int i, j;

     *conv = FALSE;
     b = FALSE;
   . sr = *zr;
     si = *zi;

#ifdef EBUG
     printf( "shift (stage 3): %g+ %gi\n", sr,si );
#endif

     for( i=1; i<=l3; i += 1 )
     {
          polyev(nn,sr,si,pr,pi,qpr,qpi,&pvr,&pvi);
          mp = cmod(pvr,pvi);
          ms = cmod(sr,si);
          twenerr = 20.0*errev(qpr,qpi,ms,mp,are,mre);

#ifdef EBUG
          printf( "mp: %g, twenerr: %g\n", mp, twenerr );
#endif
          if( mp <= twenerr )
          {
               *conv = TRUE;
               *zr = sr;
               *zi = si;
               return;
          }

          if( i > 1 )
          {
               if( b || mp < omp || relstp >= .05 )
               {
                    if( mp*.1 > omp )return;
               }
               else
               {
                    tp = relstp;
                    b = TRUE;
```

```
                    if( relstp < eta )
                         tp = eta;
                    r1 = sqrt( tp );
                    r2 = sr*(1.0+r1)-si*r1;
                    si = sr*r1+si*(1.0+r1);
                    sr = r2;
                    polyev(nn,sr,si,pr,pi,qpr,qpi,&pvr,&pvi);
                    for( j = 1; j <=5; j += 1 )
                    {
                         calct( &bool );
                         nexth( &bool );
                    }
                    omp = infin;
               }
          }

          omp = mp;
          calct( &bool );
          nexth( &bool );
          calct( &bool );
          if( !bool)
          {
               relstp = cmod(tr,ti)/cmod(sr,si);
               sr = sr+tr;
               si = si+ti;
          }
     }
     return;
}

/*
 * computes t = -p(s)/h(s).
 * bool - logical set true if h(s) is essentially zero.
 */

calct(bool)
int *bool;
{
     double hvr,hvi;
     int n;

     n = nn-1;
     polyev(n,sr,si,hr,hi,qhr,qhi,&hvr,&hvi);  /* h(s) */
     *bool = ((cmod(hvr,hvi) <= are*10.0*cmod(hr[n-1],hi[n-1]))
             ? TRUE : FALSE );

     if( *bool)
     {
          tr = 0.0;
          ti = 0.0;
     }
     else
     {
          cdivid(-pvr,-pvi,hvr,hvi,&tr,&ti);
```

```
	}
	return;
}

/*
 * calculates the next shifted h polynomial.
 * bool - logical, if .true. h(s) is essentially zero
 */

nexth(bool)
int *bool;
{
	double t1,t2;
	int n, nm1, j;

	n = nn-1;
	nm1 = n-1;
	if( *bool)
	{
		for( j = 2; j <= n; j += 1 )
		{
			hr[j-1] = qhr[j-2];
			hi[j-1] = qhi[j-2];
		}
		hr[0] = 0.0;
		hi[0] = 0.0;
		return;
	}
	else
	{
		for( j = 2; j <=n; j += 1 )
		{
			t1 = qhr[j-2];
			t2 = qhi[j-2];
			hr[j-1] = tr*t1-ti*t2+qpr[j-1];
			hi[j-1] = tr*t2*ti*t1+qpi[j-1];
		}
		hr[0] = qpr[0];
		hi[0] = qpi[0];
	}
	return;

}

/*
 * evaluates a polynomial p at s by the horner recurrence
 * placing the partial sums in q and the computed value in pv.
 */

polyev(n,sr,si,pr,pi,qr,qi,pvr,pvi)
double *pr, *pi, *qr, *qi, sr,si, *pvr, *pvi;
int n;
{
	int i;
```

```
     double t;

     qr[0] = pr[0];
     qi[0] = pi[0];
     *pvr = qr[0];
     *pvi = qi[0];
     for( i = 2; i <= n; i += 1 )
     {
          t = (*pvr)*sr-(*pvi)*si+pr[i-1];
          *pvi = (*pvr)*si+(*pvi)*sr+pi[i-1];
          *pvr = t;
          qr[i-1] = *pvr;
          qi[i-1] = *pvi;
     }
     return;
}

/*
 * bounds the error in evaluating the polynomial by the horner
 * recurrence.
 * qr,qi - the partial sums
 * ms - modulus of the point
 * mp - modulus of the polynomial value
 * are, mre - error bounds on complex addition and multiplication
 */

double errev(qr,qi,ms,mp,are,mre)
double *qr, *qi, ms,mp,are,mre;
{
     int i;
     double e;

     e = cmod(qr[0],qi[0])*mre/(are+mre);
     for( i = 1; i <= nn; i += 1 )
     {
          e = e*ms+cmod(qr[i-1],qi[i-1]);
     }
     return( e*(are+mre)-mp*mre );

}

/*
 * cauchy computes a lower bound on the moduli of the zeros of a
 * polynomial - pt is the modulus of the coefficients
 */

double cauchy(pt,q)
double *q, *pt;
{
     double x,xm,f,dx,df;
     int i, n;

     n = nn-1;
     pt[n] = -pt[n];
```

```
	/* upper estimate of bound */
	x = exp( (log(-pt[n]) - log(pt[0]))/( ( double )n) );
	if( pt[n-1] != 0.0 )
	{
		xm = -pt[n]/pt[n-1];
		if( xm<x ) x = xm;
	}

	for( f = 1.0; /* kluge */ f > 0.0; x = xm )
	{
		xm = x*0.1;
		f = pt[0];
		for( i = 2; i <=nn; i += 1 )
		{
			f = f*xm+pt[i-1];
		}
	}

	/* Newton iteration until x converges to two decimal places */
	dx = x;
	while( dabs(dx/x) > 0.005)
	{
		q[0] = pt[0];
		for( i = 2; i <=nn; i += 1 )
		{
			q[i-1] = q[i-2]*x+pt[i-1];
		}
		f = q[n];
		df = q[0];
		for( i = 2; i <= n; i += 1 )
		{
			df = df*x+q[i-1];
		}
		dx = f/df;
		x = x-dx;
	}

	return( x );

}

/*
 * returns a scale factor to multiply the coefficients of the
 * polynomial. the scaling is done to avoid overflow and to avoid
 * undetected underflow interfering with the convergence
 * criterion. the factor is a power of the base.
 * pt - modulus of the coefficients of p
 * eta,infin,smalno,base - constants describing the
 * floating point arithmetic.
 */

double scale(pt)
double *pt;
{
```

```
	double hi, lo, max, min, x, sc, l;
	int i;

	/* find largest and smallest moduli of coefficients */
	hi = sqrt(infin);
	lo = smalno/eta;
	max = 0.0;
	min = infin;
	for( i = 1; i <= nn; i += 1 )
	{
		x = pt[i-1];
		if( x > max )
			max = x;
		if( x != 0.0 && x < min )
			min = x;
	}

	/* only necessary with very large or very small components */
	if( min >= lo && max <= hi )
		return( 1.0 );
	x = lo/min;
	if( x > 1.0 )
	{
		sc = x;
		if( infin/sc > max )
			sc = 1.0;
	}
	else
	{
		sc = 1.0/(sqrt(max)*sqrt(min));
	}
	l = ceil(log(sc)/log(base));

	return( pow(base,l) );
}

/* complex division, avoiding overflow */

cdivid(ar,ai,br,bi,cr,ci)
double ar,ai,br,bi,*cr,*ci;
{
	double r,d;

	if(br != 0.0 || bi != 0.0 )
	{
		if( dabs(br) >= dabs(bi))
		{
			r = bi/br;
			d = br+r*bi;
			*cr = (ar+ai*r)/d;
			*ci = (ai-ar*r)/d;
			return;
		}
		else{
```

```
                r = br/bi;
                d = bi+r*br;
                *cr = (ar*r+ai)/d;
                *ci = (ai*r-ar)/d;
                return;
            }
        }
        else /* division by zero, c = infinity */
        {
            mcon();
            *cr = infin;
            *ci = infin;
            return;
        }

}

/* modulus of a complex number, avoiding overflow */

double cmod(r,i)
double r,i;
{
        double ar, ai;

        ar = dabs(r);
        ai = dabs(i);
        if( ar > ai )
            return( ar*sqrt(1.0+(ai/ar)*(ai/ar)));
        else
        {
            if( ar < ai )
                return( ai*sqrt(1.0+(ar/ai)*(ar/ai)));
            else
                return( ar*M_SQRT2 );
        }
}

/*
 * mcon provides machine constants used in various parts of the
 * program. the user may either set them directly or use the
 * statements below to compute them. the meaning of the four
 * constants are -
 * eta the maximum relative representation error
 * which can be described as the smallest positive
 * floating-point number such that 1.0d0 + eta is
 * greater than 1.0d0
 * infiny the largest floating point number
 * smalno the smallest positive floating-point number
 * base the base of the floating-point number system used
 * let t be the number of base-digits in each floating-point
 * number(double precision). then eta is either .5*b**(1-t)
 * or b**(1-t) depending on whether rounding or truncation
 * is used
 * let m be the largest exponent and n the smallest exponent
```

```
 * in the number system. then infiny is (1-base**(-t))*base**m
 * and smalno is base**n.
 * the values for base,t,m,n below correspond to the ibm/360.
 *
 * *NOTE* revised for IEEE floating point
 *
 */

mcon()
{
     base = 2.0;

#ifdef vax
     eta = pow(base,-54.0);
     infin = pow(base,126.0);
     smalno = 1.0/infin;
#else
     eta = pow(base,(1.0-52.0));
     infin = pow(base,1023.0);
     smalno = 1.0/infin;
#endif

#ifdef EBUG
     printf( "eta=%e, 1.0+eta=%e, infin=%e, smalno=%e\n",
             eta, 1.0+eta, infin, smalno );
#endif
     return;
}
```

22. Appendix XIV: cvaryangle.c

```
#include <stdio.h>
#include <sys/param.h>
#include <sys/types.h>
#include <sys/times.h>

#define MAX_DEGREE 50

/*
 * batch interface program for j-t method,
 * found in this directory in file "cjt.c"
 */
main( argc, argv, env )
char *argv[], *env[];
{
     long clock, times(), min, avg, max, user, nprocs;
     struct tms tb1, tb2;
     int i, degree, flflag, count, fail_count;
     double init, coeffr[MAX_DEGREE], coeffi[MAX_DEGREE],
     resulr[MAX_DEGREE], resuli[MAX_DEGREE], offset;

     scanf( "%d", &degree );
#ifdef EBUG
     fprintf( stderr, "degree: %d\n", degree );
#endif
     if( degree > MAX_DEGREE-1 || degree < 0 )
     {
          fprintf( stderr, "0<= degree <= %d", MAX_DEGREE-1 );
          exit( 1 );
     }

     for( i = 0; i <= degree; i += 1 )
     {
          scanf( " %lf, %lf", &coeffr[i], &coeffi[i] );
     }

#ifdef EBUG
     for( i=0; i <= degree; i += 1 )
     {
          fprintf( stderr, "%g, %g\n", coeffr[i], coeffi[i] );
     }
#endif
/*
 * apply the method for a sequence of angles
 */

     for( nprocs=1; nprocs <= 50; nprocs += 1 )
     {
          fail_count = count = min = max = avg = 0;
          offset = (double) 360.0/nprocs;
          for( init=3.0;
               init <= 360.0; init += offset )
```

```
		{
			count += 1;
			clock = times( &tb1 );

			cpoly( coeffr, coeffi, degree, resulr, resuli, &flflag,
			(init/180.0)*3.14159265358979323846 );

			clock = times( &tb2 ) - clock;
			user =  tb2.tms_utime - tb1.tms_utime;

			if( flflag )
			{
#ifdef FAIL
				fprintf( stderr,
					"cpoly() failed on input. Exiting.\n" );
				exit( 1 );
#else
				fail_count += 1;
#endif
			}
#ifdef VERBOSE
			printf( "\nangle: %g, time %d ticks, zeros at:\n",
				init, user );
			for( i=0; i <= degree-1; i += 1 )
			{
				printf( "%e, %e\n", resulr[i], resuli[i] );
			}
#endif
			if( min == 0 && !flflag )
				min = user;
			if( user < min && !flflag )
				min = user;
			if( user > max )
				max = user;
			avg += user;

		}
		avg = avg / count;

		printf( "nprocs: %d, max: %d, min: %d, avg: %d, fails: %d\n",
			nprocs, max, min, avg, fail_count );

	}
	exit( 0 );
}
```

23. Appendix XV: r2p.c

```
#include <stdio.h>

#define MAX_DEGREE 50
#define EOL '\n'
#define BUFSIZE 512
#define EOS '\0'

struct complex {
     double r;
     double i;
};

typedef struct complex COMPLEX;

COMPLEX tmp[MAX_DEGREE],
     roots[MAX_DEGREE],
     coeff[MAX_DEGREE];

main( argc, argv, env )
int argc;
char *argv[], *env[];
{
     FILE *fp;

     if( argc > 2 )
          usage();
     if( argc == 2 )
     {
          fp = fopen( argv[1], "r" );
          if( fp == (FILE *) NULL )
          {
               perror( argv[1] );
               exit( 1 );
          }
     }
     else
          fp = stdin;

     load_roots( fp );
     close( fp );
     compute_poly();
     print_poly();
     exit( 0 );

}

usage()
{
     fprintf( stderr, "Usage: r2p [filename]\n" );
     exit( 1 );
}
```

```
print_poly()
{
     int deg;
     register int i;

     deg = degree( coeff );
     printf( "%d\n", deg );

     for( i = 0; i <= deg; i += 1 )
     {
          printf( "%f, %f\n",  coeff[i].r, coeff[i].i );
     }

     return;
}


load_roots( file )
FILE *file;
{
     int i, getline();
     char line[BUFSIZE], *line_ptr, *get_float();

     for( i = 0;
          getline( line, BUFSIZE ) > 0;
          i += 1 )
     {
          line_ptr = line;
          line_ptr = get_float( line_ptr, &roots[i].r );
          get_float( line_ptr, &roots[i].i );

#ifdef EBUG
          printf( "load_roots: got %g+%gi\n",
                    roots[i].r, roots[i].i );
#endif
     }

     return;
}

compute_poly()
{
     int i;
     int deg;

     deg = degree( roots );

     coeff[1].r = -roots[0].r;
     coeff[1].i = -roots[0].i;
     coeff[0].r = 1.0;
     coeff[0].i = 0.0;

     for( i = 1; i <= deg; i += 1 )
     {
```

```
            roots[i].r = -roots[i].r;
            roots[i].i = -roots[i].i;
            do_root( &roots[i] );
        }

        return;
}

do_root( root )
COMPLEX *root;
{
        register int i;
        int deg;
        COMPLEX c;

        deg = degree( coeff );
        cprod( root, &coeff[deg], &c );
        tmp[deg+1].r = c.r;
        tmp[deg+1].i = c.i;
        tmp[0].r = 1.0;
        tmp[0].i = 0.0;

        for( i = 1; i <= deg; i += 1 )
        {
            cprod( root, &coeff[i-1], &c );
            csum( &coeff[i], &c, &tmp[i] );
        }

        for( i = 0; i <= deg+1; i += 1 )
            coeff[i] = tmp[i]; /* needs structure assign. */

        return;
}

cprod( a, b, c )
COMPLEX *a, *b, *c;
{
        c->r = a->r*b->r - a->i*b->i;
        c->i = a->i*b->r + a->r*b->i;
        return;
}

csum( a, b, c )
COMPLEX *a, *b, *c;
{
        c->r = a->r + b->r;
        c->i = a->i + b->i;
        return;
}

int
degree( poly )
COMPLEX *poly;
{
```

```
	register int i;
	int deg;

	for( i = 0; i < MAX_DEGREE; i += 1 )
		if( poly[i].r !=0.0 || poly[i].i != 0.0 )
			deg = i;

#ifdef EBUG
	printf( "degree: returning %d\n", deg );
#endif

	return( deg );
}

/*
 * safe "gets()"
 * leaves terminating newline
 * always returns null-terminated string.
 */

int
getline( buf, size )
char *buf;
int size;
{
	int c, counter;

	if( buf == (char *) NULL )
		return( 0 );

	for( counter = 0; counter < size-1; )
	{
		c = getchar();
		if( c == EOF )
			break;
		buf[counter++] = c;
		if( c == EOL )
			break;
	}
	buf[counter] = EOS;
	return( counter );
}

char *
get_float( str, dub )
char *str;
double *dub;
{
	char copy[BUFSIZE], *ptr;

	if( str == (char *) NULL )
		return( (char *) NULL );

	while( *str != EOS )
```

```
        {
            if( *str >= '0' && *str <= '9' )
                break;
            if( (*str == '+' || *str == '-' || *str == '.') &&
                (*(str+1) >= '0' && *(str+1) <= '9' ) )
                break;
            ++str;
        }

        ptr = copy;

        while( *str != EOS )
        {
            if( (*str < '0' || *str > '9' )
                && *str != '+'
                && *str != '-'
                && *str != '.'
                && *str != 'e' )
            {
                *ptr = EOS;
#ifdef EBUG
                printf( "copy was: %s\n", copy );
#endif

                sscanf( copy, "%lf", dub );
                return( str );
            }
            *ptr = *str;
            ++ptr;
            ++str;
        }

        return( str );

}
```

24. Appendix XVI: cmach.c

```
#include <stdio.h>
#include <sys/param.h>
#include <sys/types.h>
#include <sys/times.h>
#include <sys/signal.h>
#include <sys/time.h>

#define MILLION 1000000
#define MAX_DEGREE 50
#define FILE_NAME_LEN 128
char sema_file[FILE_NAME_LEN];

/*
 * "parallel" batch interface program for j-t method,
 * found in this directory in file "cjt.c"
 */
main( argc, argv, env )
char *argv[], *env[];
{
     int catcher();
     long clock, user, nprocs;
     struct tms tb1, tb2;
     int i, fd, degree, flflag, count, pid, ppid, status;
     double init, coeffr[MAX_DEGREE], coeffi[MAX_DEGREE],
     resulr[MAX_DEGREE], resuli[MAX_DEGREE], offset;
#ifdef SYSTEM_FIVE
     long times();
#else
     struct timeval tv1, tv2;
     struct timezone tz;
#endif

     scanf( "%d", &degree );

     if( degree > MAX_DEGREE-1 || degree < 0 )
     {
          fprintf( stderr, "0<= degree <= %d", MAX_DEGREE-1 );
          exit( 1 );
     }

     for( i = 0; i <= degree; i += 1 )
     {
          scanf( " %lf, %lf", &coeffr[i], &coeffi[i] );
     }

     /*
      * apply the method for a sequence of angles
      */

     pid = fork();
```

```
	switch( pid )
	{
	case -1:
		fprintf( stderr, "Can't fork; exiting!\n" );
		exit( 1 );

	case 0:
		break;

	default:
		while ( wait( &status ) >= 0 )
			;
		_exit( 1 );
	}

	for( nprocs=1; nprocs <= 6; nprocs += 1 )
	{
#ifdef SYSTEM_FIVE
		clock = times( &tbl );
#else
		gettimeofday( &tvl, &tz );
#endif
		setpgrp();
		ppid = getpid();
		get_sema();
		signal( SIGTERM, catcher );
		offset = (double) 360.0/nprocs;
		for( init=3.0;
			init <= 360.0; init += offset )
		{
			pid = fork();

			switch( pid )
			{
			case -1:
				fprintf( stderr, "Can't fork, exiting!\n" );
				exit( 1 );

			case 0:
				cpoly( coeffr, coeffi, degree,
				 resulr, resuli, &flflag,
				 (init/180.0)*3.14159265358979323846 );
				if( flflag )
				{
					_exit( 1 );
				}

				lock_sema();
				signal( SIGTERM, SIG_IGN );
				kill(0, SIGTERM );
				break;

			default:
				break;
```

```
                }
                if( pid == 0 )
                     break;
          }
          if( pid != 0 )
          {
               while( wait( &status ) > 0 )
                     ;
               free_sema();
          }

#ifdef SYSTEM_FIVE
          clock = times( &tb2 ) - clock;
          printf( "nprocs: %d, time: %d ticks\n", nprocs, clock );
#else
          gettimeofday( &tv2, &tz );
          /* adjust as per manual page */
          if( tv1.tv_usec > tv2.tv_usec )
          {
               tv2.tv_usec += MILLION;
               tv2.tv_sec -= 1;
          }
          clock = MILLION*(tv2.tv_sec-tv1.tv_sec)+
               (tv2.tv_usec-tv1.tv_usec);
          printf( "nprocs: %d, time: %d usec\n", nprocs, clock );
#endif
     }
     exit( 0 );
}

#ifdef HPUX

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int Semid;
struct sembuf Sops[1];
ushort Sinit[1];

get_sema()
{
     key_t k;
     int ret;

     sprintf( sema_file, "/tmp/CMACH%d", getpid() );
     k = ftok( sema_file, 'A' );
     ret = semget( k, 1, IPC_CREAT | 0600 );
     if( ret < 0 )
     {
          fprintf( stderr, "get of semaphore failed. Exiting.\n" );
          exit( 1 );
     }
     Semid = ret;
```

```
	Sinit[0] = 1;
	if( semctl( Semid, 0, SETALL, Sinit ) < 0 )
	{
		fprintf( stderr, "set of semaphore failed. Exiting.\n" );
		exit( 1 );
	}

}

lock_sema()
{

	Sops[0].sem_num = 0;
	Sops[0].sem_op = -1;
	Sops[0].sem_flg = IPC_NOWAIT;

	if( semop( Semid, Sops, 1 ) < 0 )
		_exit( 1 );
	else
		return;
}

free_sema()
{
	semctl( Semid, 0, IPC_RMID, Sinit );
	return;
}

#else

get_sema()
{
	int fd;

	sprintf( sema_file, "/tmp/CMACH%d", getpid() );
	if( (fd=creat(sema_file,0)) < 0 )
	{
		fprintf( stderr, "Can't creat %s; exiting!\n",
			sema_file );
		exit( 1 );
	}
	else
		close( fd );

	return;
}

free_sema()
{
	lock_sema();
}

lock_sema()
{
```

```
        if( unlink( sema_file ) == -1 )
        {
            exit( 0 );
        }
        return;
    }

    #endif

    int catcher( sig )
    {
        if( sig == SIGTERM )
        {
            signal( SIGTERM, catcher );
#ifdef EBUG
            fprintf( stderr, "pid %d, caught SIGTERM\n", getpid() );
#endif
            exit( 1 );
        }
        else
            return;
    }
```

25. Appendix XVII: Extremal exploitation of randomness

```
rand_search.c:
#include <stdio.h>
#include <math.h>
#include <sys/types.h>
#include <sys/times.h>
#include <sys/time.h>

#define MILLION 1000000
#define TRUE 1
#define FALSE 0

#ifdef EBUG
#define PERM_FILE "/tmp/Permutation"
#endif

#define MAX_SIZE 100000

int S[MAX_SIZE],
     Ticks[MAX_SIZE],
     P[MAX_SIZE],
     SSize = -1, PSize = -1;

main( argc, argv )
int argc;
char *argv[];
{
     double total, scaler, dSize, floor(), drand48();
     register int i, j, k;
     int min, max;
     long clock;
     struct timeval tv1;
     struct timezone tz;

     while( argc > 1 )
     {
          if( argv[argc-1][0] != '-' )
               usage();
          switch( argv[argc-1][1] )
          {
          case 's':
               PSize = atoi( &argv[argc-1][2] );
               if( PSize < 1 || PSize > MAX_SIZE )
                    usage();
               break;

          case 'c':
               SSize = atoi( &argv[argc-1][2] );
               if( SSize < 1 || SSize > MAX_SIZE )
                    usage();
               break;
```

```
            default:
                usage();
            }
            argc -= 1;
        }

        if( PSize == -1 )
            PSize = MAX_SIZE;
        if( SSize == -1 )
            SSize = MAX_SIZE;

        dSize = (double) PSize;


        gettimeofday( &tv1, &tz );
        clock = MILLION*tv1.tv_sec+tv1.tv_usec;
        srand48( clock );

        for( i = 0; i < PSize; i += 1 )
            P[i] = i;

        /* this might have to be done better */
        for( i = 0; i < PSize; i += 1 )
        {
            k = (int) floor( drand48()*dSize );
            j = P[i]; P[i] = P[k]; P[k] = j;
        }

#ifdef EBUG
        {
            FILE *fp;

            fp = fopen( PERM_FILE, "w" );
            if( fp == (FILE *) NULL )
            {
                fprintf( stderr,"Can't open %s for writing. Exiting\n",
                    PERM_FILE );
                exit( 1 );
            }
            fprintf( fp, "Elements: %d\n", PSize );
            for( i = 0; i < PSize; i += 1 )
                fprintf( fp, "%d%c", P[i], (i%10)?',':'\n' );
            fclose( fp );
        }
#endif

        k = (int) floor( drand48()*dSize );

        for( i = 0; i < SSize; i += 1 )
        {
            S[i] = (int) floor( drand48()*dSize );

            gettimeofday( &tv1, &tz );
            clock = MILLION*tv1.tv_sec+tv1.tv_usec;
```

```
            Ticks[i] = clock;

            if( P[find_index( k, S[i] )] != k )
                  fprintf( stderr, "find_index failed. continuing.\n" );

            gettimeofday( &tv1, &tz );
            clock = MILLION*tv1.tv_sec+tv1.tv_usec;
            Ticks[i] = clock - Ticks[i];
      }

      /* dump statistics */
      scaler = 1.0/((double) MILLION);
      min = max = Ticks[0];
      /* we pre-scale Ticks to prevent overflow */
      total = scaler*Ticks[0];

      for( i = 1; i < SSize; i += 1 )
      {
            if( Ticks[i] < min )
                  min = Ticks[i];
            if( Ticks[i] > max )
                  max = Ticks[i];
            total += scaler*Ticks[i];
      }

      printf( "min: %g sec, max: %g sec, avg: %g sec.\n",
            scaler* (double) min,
            scaler* (double) max,
            total / (double) SSize );

      exit( 0 );
}

int
find_index( element, start )
int element, start;
{
      int i, offset, halfSize;

      halfSize = PSize/2;

      for( offset = 0; offset <= halfSize; offset += 1 )
      {
            i = start - offset;
            if( i < 0 )
                  i += PSize;
            if( equal( P[i], element ) )
                  return( i );
            i = start + offset;
            if( i >= PSize )
                  i -= PSize;
            if( equal( P[i], element ) )
                  return( i );
      }
```

```
     return( -1 );

}

int
equal( i, j )
int i, j;
{
     if( i == j )
          return( 1 );
     return( 0 );
}

usage()
{
     fprintf( stderr, "Usage: rand_search [-ssize] [-ccount]\n" );
     fprintf( stderr, "size & count <= %d\n", MAX_SIZE );
     exit( 1 );
}

script:
for i in 1 2 5 10 20 50 100 200 500 1000 5000 10000 50000 100000
do
     rand_search -c${i}
done

results:
min: 0.361704 sec, max: 0.361704 sec, avg: 0.361704 sec.
min: 0.038728 sec, max: 0.158708 sec, avg: 0.098718 sec.
min: 0.019244 sec, max: 0.606028 sec, avg: 0.385311 sec.
min: 0.022124 sec, max: 0.68992 sec, avg: 0.249112 sec.
min: 0.020452 sec, max: 0.698092 sec, avg: 0.456766 sec.
min: 0.031444 sec, max: 0.70512 sec, avg: 0.330535 sec.
min: 0.00248 sec, max: 0.71294 sec, avg: 0.330925 sec.
min: 0.00034 sec, max: 0.739772 sec, avg: 0.3693 sec.
min: 0.002532 sec, max: 1.18184 sec, avg: 0.383545 sec.
min: 0.001124 sec, max: 0.710796 sec, avg: 0.35026 sec.
min: 0.00024 sec, max: 2.53285 sec, avg: 0.356609 sec.
min: 0.000164 sec, max: 2.85144 sec, avg: 0.367076 sec.
min: 0.000176 sec, max: 8.0722 sec, avg: 0.371924 sec.
min: 0.00012 sec, max: 2.39245 sec, avg: 0.358988 sec.
```

26. Biography

Jonathan Michael Smith was born on October 10, 1959 in Taunton, Massachusetts, where he attended elementary and secondary school. He received his Bachelor of Arts degree in Mathematics, Magna Cum Laude, from Boston College, Chestnut Hill, Massachusetts in May, 1981. Jonathan's senior Honors thesis in number theory was on ``Full Period Primes,'' which are prime numbers whose unitary fraction has the longest possible repeating decimal expansion.

After graduating, Jonathan was employed by Bell Telephone Laboratories. From September 1982 until May 1983, he was supported in his studies towards a Master of Science in Computer Science in the Laboratories' One Year on Campus graduate studies program. Columbia University granted the M.S. degree in May of 1983. On January 1 of 1984, Jonathan became an employee of Bell Communications Research, Inc. as a result of the AT&T divestiture. He returned to Columbia University in September of 1984 in order to pursue a Doctor of Philosophy degree in Computer Science.