1987

A computer simulation of processor scheduling in UNIX 4.2BSD

Michael D. Grossman

Rochester Institute of Technology

School of Computer Science and Technology

A Computer Simulation of Processor Scheduling in UNIX 4.2BSD

by

Michael D. Grossman

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

Professor James R. Carbin

Professor Warren R. Carithers

Professor Andrew Kitchen

1.2 Abstract

This project is a study of the processor scheduling system in UNIX 4.2BSD. This study involved a computer simulation of the processor scheduling system. The preliminary work for the simulation included choosing a system, choosing and running a set of test processes on that system, gathering statistics from these runs, and constructing a model of the scheduling system. The model was then tuned to perform like the real system by introducing overhead into the model. The overhead was added using several variables in the model. Tuning consisted of adjusting the values of these variables until the performance of the model was as close as possible to that of the real system. Experiments were performed on the model consisting of a rescheduling experiment that examined the handling of compute-bound processes by the scheduler and several experiments that study the effects of modifications to the scheduler.

1.3 Computing Review Subject Codes

D.4.8 [Operating Systems]: Performance-simulation

D.4.1 [Operating Systems]: Process Management-scheduling

1.4 Table of Contents

2. Proposal

The text of the proposal is found in Appendix A.

Modifications to the Proposal

In the proposal, the exact formula for the computation of the priority of a process was given, and it was assumed that the computation of priorities in the model would be exactly the same as in the real system. However, it was decided to ignore the memory usage factor. In the real system, if the amount of memory in use is greater than the amount considered to be desirable and the resident set size of the process that is having its priority set is greater than it should be at this point in its execution, the priority of the process is lowered by eight.

Upon reflection, the resident set size of a process at any point in its execution could not be determined nor could it even be estimated. Although the command "ps axl" provided the resident set size, it was not that helpful since that command was run only once every two seconds. Two seconds is equal to two hundred system clock ticks. Therefore, the resident set size could easily change significantly between runs of "ps axl", and there is no way to obtain this information. Thus, the statistics could not be used even to formulate an educated guess.

The other difference between the proposed and actual models is that the model was written in C rather than in GPSS. C was chosen because it is more readily available and because GPSS offered no real advantage for this project over C. This particular system does not require the features of GPSS to model it. It can be done in C with only a little extra effort.

3. Introduction and Background

The 4.2BSD version of UNIX is known to suffer from a performance problem in that response times are often rather slow on a VAX-11 series machine. Besides the general sluggishness of the system, there are times that executing a command suddenly takes much longer than before with no apparent increase in the load level.

This project looks into one potential problem area: The processor scheduling algorithm. The first part of the task was to create a computer model of the processor scheduling system in UNIX 4.2BSD and to perform simulation runs of the model using statistics from the real system as input to the model. A model of the scheduler was constructed and tuned until it reflected as closely as possible the performance of the real system. The performance was judged in terms of the response times of the members of the family of processes that was described in the proposal. The second part of the task involved performing experiments on the model to investigate whether the scheduler treats I/O-bound processes fairly, and whether any simple modifications to the scheduler would improve its performance in terms of either better response time or more uniform response times for members of the family of processes.

4. Collection of Statistics

4.1 Run of Daemon

As was described in the proposal, a daemon was run that fired up the family of processes each time the load level of the system reached one of several values. The daemon was designed to begin execution of the family at load levels of two, four, six, and eight. Unfortunately, the system being observed generally had only a light or moderate load on it. Consequently, only runs at a load level of two were able to be observed. One hundred eleven runs of the daemon were made.

The commands that made up that part of the family of processes designated as "user processes" were:

1. egrep Grossman /etc/passwd
2. w
3. ps g
4. nroff -man /usr/man/man1/cc.1

5. sh /u/staff/gros/thesis/dmn/exscript

6. pi /u/staff/gros/thesis/dmn/llist.p

7. pc /u/staff/gros/thesis/dmn/llist.p

8. cc /u/staff/gros/thesis/dmn/pdmn.c

"exscript" is a shell script that uses the "ex" editor to create a file. "llist.p" is a Pascal program that does some simple linked list operations. "pdmn.c" is the daemon used to fire up the family of processes. It should be noted that in the project, the statistics from the "ps" command were unable to be used because it could not be determined from the system statistics which run of "ps" was the user process run and which were the statistics gathering runs of "ps axl" and "ps axv". This problem was recognized while the daemon and processes were being designed, and an attempt was made to avoid it by creating symbolic links to the "ps" command called psl and psv. psl was the "ps" command called when running "ps axl" while psv was called to run "ps axv". Unfortunately, the system statistics did not record the name of the symbolic link as the command name but rather the real name of the command, namely ps.

4.2 Statistics that were Ignored

After careful consideration, it was decided that the statistics from the "vmstat", "ps axl", and "ps axv" commands would not be used for the most part. The ps axl command was run to obtain the nice setting, the priority, and the resident set size of each process in the system at two second intervals. The "ps axv" command was run at two second intervals to record the number of pages each process had read in. The "vmstat" command, run at five second intervals, was used to obtain the size of the memory free list. It was felt that the "ps axv" statistics, while useful for general paging activity, would be difficult to incorporate into the model. For each page transferred, an estimate would have to be made on how long it actually took to perform the transfer of the page from disk. Since the only information available on the performance of the disk was the

data concerning seek time, latency, and disk transfer rate that were mentioned by the manufacturer in his documentation describing the disk drive (a DEC RA81), it would be quite difficult to get very accurate figures for the particular system as the members of the family of processes were running. Thus knowledge about the amount of paging would not be that helpful since the transfer rate for each page would have to have been determined randomly and could potentially vary significantly depending on disk system activity at any point in time. It was decided that just assuming that paging was included as part of the total overhead of the processes that were not family members (i.e., those processes referred to in this paper as background processes) and tuning the model accordingly would yield comparable results and be much simpler to implement.

The other major use that was contemplated for these statistics was to determine whether a particular process was using too much memory and had to have its priority reduced as is provided for in the computation of the priority in the real system. The problem with this intended use was that these statistics were derived from snapshots of the system taken at two or five second intervals. In a UNIX 4.2BSD system running on a VAX-11/750, clock interrupts occur every hundredth of a second. A process is allowed to run for at most one tenth of a second before it must relinquish the CPU. Thus, intervals of two or five seconds are relatively large. The resident set size of a process or the amount of free memory available in the system when the priority of a process is recomputed at a clock interrupt could be very different from the figure recorded by the last run of "ps axl" or "vmstat" which might have occurred several hundred time units before the clock interrupt. Therefore, these statistics were not used, and this part of the computation of the priority was ignored.

4.3 Statistics Used in the Simulation

The simulation runs used primarily the statistics gathered by system accounting on the real system for input. The data of interest from the system accounting files were the

time each process began execution, its elapsed time in the system, the amount of CPU time it consumed, and the number of I/O blocks read from or written to disk by that process.

The CPU time recorded by the 4.2BSD accounting system for a completing process is in seconds even though the kernel records the time in hundredths of a second. A unit of seconds was unacceptably large for CPU time since many processes consume less than one second of CPU time. Therefore, a minor modification was made to the accounting system in the kernel of the system being modelled so that it reported CPU time in hundredths of a second.

The only statistic used in the model that did not come from the system accounting files was the priority of processes already running when the first user process of the family of processes began execution. This was taken from the first run of the "ps axl" command after the first user process in the family had started in every run of the family.

In addition to the statistics mentioned above, the total number of processes that ran during each run of the family was derived from the system accounting statistics and recorded.

5. Organization of Input

In the further discussion of the project any reference to the members of the family of processes should be taken to mean only the user processes that were members of the family of processes. The statistics gathering processes run by the daemon are identified as such. This change is warranted because from this point on most of the discussion centers on the model and the simulation runs which treat the statistics gathering processes as ordinary background processes.

From the accounting files for the system, those statistics associated with all processes that ran while the family ran were extracted. For each run of the daemon, one file containing these data was created. The raw data were then converted into actual

files that the model could use for input in the following manner. All of the processes described in the raw accounting files created for each run of the daemon were divided into three categories. The first group consisted of those processes that began executing before any member of the family began but did not complete until after the first member of the family of processes had started executing. The second group included those processes that began executing after the first member of the family had begun. The third group was made up of the members of the family of processes. For the first two groups, one file was created for each of the following statistics:

1. The time that the process began execution.
2. The amount of CPU time it consumed.
3. The amount of time that elapsed from when it began until it completed.
4. The number of I/O blocks that were transferred from or to disk.

The organization of the data for the members of the family of processes was different in that individual files for each of these statistics were created for each family member. The format of all of these files was the same. The first three lines of the file contained the minimum, maximum, and average values of the statistic. The succeeding lines each contained a value of the statistic, usually rounded, and the number of processes for which this value corresponded to the value given in the system statistics after it was rounded. The number of significant digits in different data files varied depending on the size and range of the values for the particular statistic.

There were three other data files containing other information, formatted differently than previously described. The first such file contained the number of processes in each run that started before but completed after the first family member began. The second file contained the number of processes that began after the first family member had begun but before the last family member had completed in each run of the family. The format of these files was as follows. The first three lines contained the

minimum, maximum, and average number of such processes for the one hundred eleven runs of the family. The lines following these data contained the number of the run of the family and the number of processes in that run that fit the requirements for inclusion in the particular file.

The third file contained data concerning the priorities that those processes that had begun before the first family member started had at the start of the execution of the first family member. Although such data was impossible to obtain at the exact instant of the beginning of the execution of the family, the priorities of the processes in question were found as soon as possible after the start of the first family member. They were extracted from the first entry in each file created by the "ps axl" command. There was one such file for each run of the family. The format of the data file created for the simulation that contained this information was similar to that of the files previously described. The first three lines contained the minimum, maximum, and average priorities for all runs. Each of the next lines corresponded to one run of the family of processes and contained the number of the run, the maximum priority, the minimum priority, and the average priority for all processes in that run that had begun before the family.

6. Design of the Model

6.1 Introduction

The way in which processor scheduling is performed in UNIX 4.2BSD is described in the proposal. The intention of the model was not to duplicate the scheduler but rather to mimic the function of the scheduler to the point that the peformance of the model was a good approximation of that of the real system. Those parts of the operating system which are external to the scheduler but which affect its operation were represented by exogenous variables. In particular, disk I/O, terminal I/O, and system housekeeping were treated in this manner. The exact data concerning these factors were not available from the system; and, therefore, approximations had to be made. This matter is discussed in

much greater detail in Section 7.2.2. Since the simulation was designed to be a rough approximation of the working of the real system, each experiment should show the tendencies that the real system would exhibit under the circumstances in the experiment.

6.2 The Major Data Structures

Each process in the simulation was represented by a proc structure that contained all of the information necessary for the process to advance in the simulation. At any given time, the proc structure representing some process was either on one of three queues or was the currently running process. The first of the three queues was the events chain, which was ordered in terms of increasing system (re)entry time contained in the proc structure field "setime". A process waited on this queue until the time specified in the "setime" field of its proc structure was reached. Its proc structure was then removed from the events chain and placed on the run queue where it waited for the CPU. The run queue was ordered on process priority. Although the real system has thirty-two queues, the model had only one queue, but the action of a thirty-two queue arrangement was simulated. This was accomplished in the model as in the real system by dividing the priority by four to obtain the number of the queue on which the process should be placed. This mechanism was implemented in the model in such a way that a process to be enqueued on the run queue was always placed after any other processes already on the run queue that had the same queue number. Thus the newly enqueued process found itself at the end of the group of processes with the same queue number just as it would in the real system. As was mentioned in the proposal, the mechanism of assigning higher priorities (i.e., zero to forty-nine) was not implemented in the model. Therefore, only queues twelve through thirty-one were found in the model.

When a process was given the CPU, it was dequeued from the run queue. A special pointer variable pointed to it, and it was not on any queue.

The third queue was a special queue that was used only by members of the family of processes. All family members were placed on this queue by the system initialization routines and waited there until the time for them to begin execution had arrived. At this time they were allowed into the model. They never reentered this queue. Its purpose was only to synchronize the family members so that they ran sequentially as in the real system. This eliminated the need to try to tune the model so that the family members ran sequentially while trying to tune it such that the response times of the family members were reasonable. Thus the tuning process was somewhat simplified.

6.3 General Design of the Model

The basic design of the model was quite straightforward. The initialization routines initialized the data structures and variables and generated the set of processes for the simulation run. Each background process was placed on the events chain, and each member of the family of processes was placed on the special initial queue for family members.

The actual simulation run was performed by a loop in the main function. Time was measured in hardware clock ticks. The unit of time in the simulation was, therefore, one hundredth of a second. This unit was chosen because it is the smallest significant time unit in the process scheduling system.

In every iteration of the loop, the clock routine was first called. This function first incremented the system clock by one time unit and then placed on the run queue any processes in the other two queues that were allowed to be placed on the run queue at this time. The clock routine either performed certain actions that were to occur at this time or set flags to allow for their occurrence. It set a flag to try to get a process from the run queue if none was currently executing. It also set flags if there was a currently executing process and some event involving this process was to occur at this time. Additionally, the clock function recomputed the priorities of all processes in the system

at the proper time and lowered the priority of the currently executing process when required by the scheduling algorithm. The rest of the loop was concerned with carrying out all events for which a flag had been set by the clock routine. This loop began executing at the start of the simulation run and stopped after the last member of the family of processes had exited.

After the run had completed, various statistics were printed.

This is a very simple description of the functioning of the model. A more detailed description follows after the block charts.

6.4 Block Charts

The block charts on the following pages show the functions that each function calls to perform its task. The order in which the functions are called is as shown from left to right. The purpose of these charts is to give a pictorial description of the model to help understand its various parts and how they work together.

Table 6.4.1

Breakdown of Function Calls by "main" to the Major Components of the Simulator

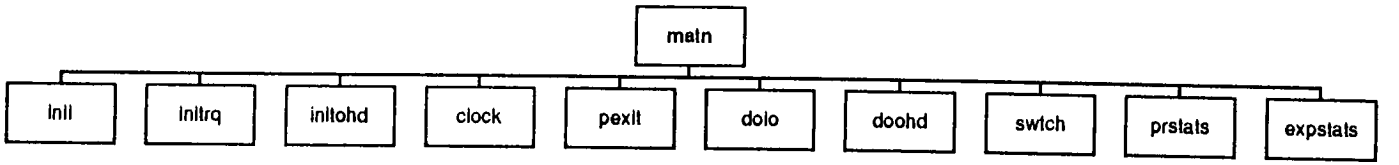


Table 6.4.2

Hierarchy of Function Calls Initiated by the Process Initialization Routine (init)

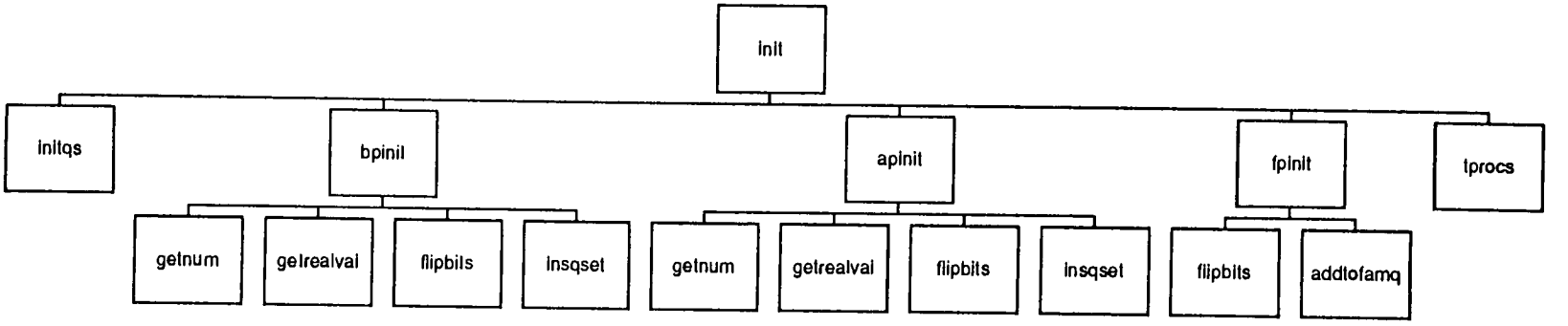


Table 6.4.3

Hierarchy of Function Calls Initiated by the Special Process Initialization Routine (tprocs)

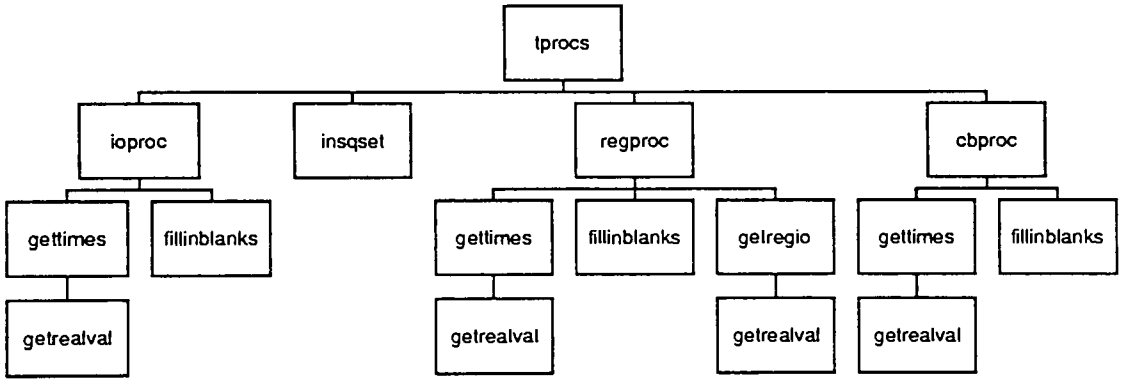
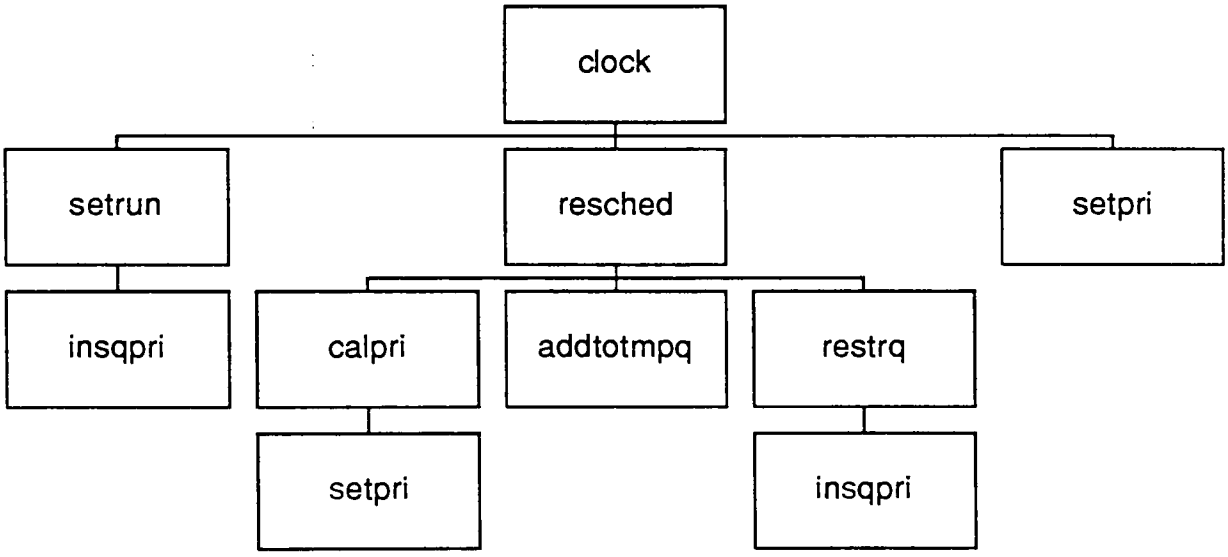


Table 6.4.4

Hierarchy of Function Calls in the Clock Routine



6.5 Detailed Description of the Model

6.5.1 The "main" Function and the Initialization Routines

The "main" function first prompted the user for the type of run he wanted and for other pertinent information. Main then called init which did the basic system initialization. Init called initqs which initialized the queue structures to be used in the simulation. Init then seeded the random number generator with a seed provided by the user and called three functions to create the three sets of processes mentioned previously for this run. These functions are bpinit, apinit, and fpinit which created the group of processes that began before the family, the group of processes that began after the family, and the family of processes itself respectively. They are all basically similar, and the general algorithm used by each one follows written in C-type pseudo-code.

pinit:

```
Get number of processes of this type;  /*Not in fpinit*/
for each process { /* for #1*/
        Get a proc structure;
        open (begin times file);
        read past minimum, maximum, and average;
        Get random number to be index into file;
        Find corresponding value of begin time variable;
        Get actual value by interpolation;
        close (begin times file);
        open (elapsed times file);
        read past minimum, maximum, and average;
        Get random number to be index into file;
        Find corresponding value of elapsed time variable;
        Get actual value by interpolation;
        close (elapsed times file);
        open (CPU times file);
        read past min., max., and average;
        Get random number to be index into file;
        Find corresponding value of CPU time;
        Get actual value by interpolation;
        close (CPU times file);

/* The I/O blocks determination is a little different.  Try to get a number of
I/O blocks such that can be processed in the given elapsed time along with the
CPU time found above.*/
```

```
        open (I/O blocks file);
        while (not found) { /* while #1 - have not yet found good value */
                read past min., max., and average;
                Get random number to be index into file;
                Find corresponding value for I/O blocks variable;
                Get actual value of variable by interpolation;
                if (number of I/O blocks reasonable) {
                    Set found = TRUE;
                } /* End if */
                else {
                    Set attempts = attempts + 1;
                } /* End else */
                if ((not found) and (attempts > = 25)) { /* if #1 - cannot get good
                    I/O value */
                    while (number of blocks not reasonable) { /* while #2*/
                        Reduce CPU time;
                    if (CPU time < = 0) { /* if #2 */
                            Set CPU time = 5;
                            Reduce number of I/O blocks;
                        }/* End if #2 */
                        } /* End while #2 */
                        Set found = TRUE;
                        Set new CPU time in proc structure;
                } /* End if #1 */
        } /*  End while #1 */
        Set I/O blocks value in proc structure;
/* Determine interval between I/O operations for this process.
    Assume one I/O operation every two ticks is the most frequent for the average
    process. */

        if (number of I/O blocks > 0) { /* if #1 */
                        if ((cpb = (CPU time)/(I/O blocks)) >= 2) /* if #2 */
                            I/O interval = cpb rounded to nearest integer;
                            Next I/O operation set to occur in "I/O internal"
                                                        ticks;
                            Number of blocks per I/O operation = 1;
                        } /* End if #2 */
                        else { /*If not normal process - i.e., less than 2 ticks
                        per I/O operation if calculated as in "if" condition. */
                            Number of blocks per I/O operation = number of
                            blocks required per transfer such that an I/O
                            operation is done at most every other tick;
                            Set I/O interval and time for next (i.e., the first)
                            I/O operation as above in if clause;
                        } /* End else */
        } * End if #1 */
        else { /* Process has been assigned 0 I/O blocks */
                Set I/O interval, time for first I/O operation, and number of
                blocks per transfer to ridiculous values such that no I/O will be
                done in the simulation;
        } * End else */
        Fill in remaining fields in proc structure;
```

```
            Put proc struct on events chain;
        } /* End for #1 */
    / *End pinit */
```

This was the general format of the process initialization routines. In particular, bpinit had to be somewhat different since the processes it created had to be set up to appear as though they were already running at system startup time. Each such process had to appear as if it had begun at the begin time determined for it, which will always be before the start of the simulation. Thus the CPU time and elapsed time for such a process during the simulation run were a fraction of the actual times determined randomly in bpinit. The procedure to find the fractions of these values for the simulation involved first examining the elapsed time to make sure that, given its begin time, the process will actually run during the simulation. If the elapsed time found would cause the process to complete before the simulation began, the process of finding an elapsed was repeated until a proper value was found. If a good value was not found after fifty tries, the process was just assigned the first value obtained and was allowed to run for that amount of time in the simulation. That is, the begin time was adjusted to time zero. The fraction of CPU time that corresponded to the fraction of elapsed time occurring after the start of the simulation was then determined and entered in the proc structure as the CPU time. Finally, the number of I/O blocks was treated in the same way. After finding a value for the total number of disk I/O blocks transferred for a process, only that fraction of the total corresponding to the ratio of the actual time the process ran in the simulation to its total elapsed time was used. After all of these values had been found, bpinit determined whether the process would be runnable or blocked for I/O at the start of the simulation run. If it would be blocked for I/O at the start of the simulation, its system entry time, which is the simulated time at which it is scheduled to enter the system, was determined to be the time necesssary for its present disk I/O to complete. A second requirement for such a process was that it had to be assigned a value for its

priority that reflected the fact that it had already begun executing. This was accomplished by randomly choosing a run of the real system and using the number of the run as an index into the file of priorities described previously in section five. The minimum and maximum values of the priorities for that run were read in, and a random value between them was chosen to be the starting priority of the process in the simulation.

The function apinit operated just as the generic initialization routine given above.

The fpinit function was also fairly similar, but it differed from the generalized routine in three ways. First, in the other two functions, the values obtained from the data files for begin time, elapsed time, CPU time, and number of I/O blocks were interpolated to get the actual value used since the increments between successive values in the data files for the other types of processes are all greater than one. However, in the data files for the family members, all such increments are one. Thus interpolation was not necessary. The second difference was that for each family member, its process structure carried information about paging in much the same way as I/O information was kept in all proc structures. This information was found by modifying the UNIX "time" command to report the number of pages read in when each member of the family of processes was run by it. The information obtained was stored in an array. The third difference between fpinit and the other process creation functions was that for each process created by fpinit a determination was made of a quantity used to control the frequency of overhead operations carried out by all other processes that run while this process was running. This quantity was derived from the values held in an array named "famohd". The optimal value for each entry in the array was found in the process of tuning the model described later.

The last aspect of the initialization routines that must be described is exactly how a value for a variable was obtained from the corresponding data file. As was mentioned in the description of these files, the format of each file is the minimum, maximum, and

average values of the quantity occupy the first three lines. Each of the remaining lines contain a rounded value of the quantity and the number of processes in the runs of the real system for which this value is the closest to the value found for this statistic in the real system. A number was randomly chosen between one and the total number of processes represented in the file. The minimum, maximum, and average values were skipped. Each successive line following these statistics was read in and a variable was incremented by the number of processes on that line until the value of the variable was greater then or equal to the randomly determined number. Then the corresponding value of the variable in the file was taken by the function. The actual value to be used in the simulation was found by an interpolation performed by the function "getrealval".

After the regular process initialization was completed, the initialization of several special processes occured if the simulation run was a run of the rescheduling experiment. If it was not, no special processes were created. These processes were different in that they all started at the beginning of the simulation and finished after the last member of the family of processes had completed. In such a case, the simulation ended after all of these processes had exited. The processes were created and their proc strutures were initialized by the function tprocs. The procedure for creating and initializing these processes was much the same as that for the normal processes. Beside being given certain values to cause them to start at the beginning of the simulation and execute until the end of the simulation, certain variables in their proc structure may have been assigned values to make them behave in a certain desired fashion. For example, a totally compute-bound process could be created by placing in the I/O fields of the proc structure values that would prevent the process from doing any I/O whatsoever. A full discussion of the nature and purpose of these processes will appear later in the discussion of the rescheduling experiment.

The rest of the main function consists of the code that actually performs the simulation of the system. This code is found mainly in one loop. The simulation does one iteration of the loop for every tick of the simulated clock. The pseudo-code for this loop and the rest of the main function follows.

```
while (not end of simulation){
    clock (); /* call clock routine. */
    if (exit flag set and there is a current process){
        pexit(); /* Allow current process to exit. */
    }
    if (ioflag or page fault flag set){
        doio (); /* Perform requested service. */
    }
    if (overhead flag set){
        doohd(); /* Perform overhead operation. */
    }
    if (switch processes flag set){
        swtch(); /* switch currently running process
                    with one on the run queue */
    }
}   /* End while */
Print statistics for family members;
if (rescheduling run){
    Print special statistics for this experiment;
}
```

6.5.2 The Clock Routine

The clock routine was the heart of the entire program. It updated the simulation clock, transferred processes from the events chain and the family of processes queue to the run queue, performed the recomputation of priorities of all processes at the proper times, reduced the priority of the running process when required, and set flags to inform the caller (i.e., the main function) to switch the CPU to a new process, to do an I/O or overhead operation, to perform paging, or to signal that the current process was to exit. There are three versions of the clock routine which differ only slightly. These include the standard clock function, the one for the rescheduling experiment, and the one used in the experiment examining the effect of changing the number of queues. Below is an outline of the general clock routine in pseudo-code.

Clock:

```
Increment system clock by 1 unit; /* clock represented by variable "systime"
*/
If(systime is multiple of 500) { /* 500 ticks = 5 seconds */
        Adjust load level using formula found in UNIX code;
} /* End if */
while(not at end of events chain) {
        Check system (re)entry time of process on events chain;
        if ((re)entry time < = systime) {
                Put process on run queue;
        } /* End if */
        else {
                End search of events chain; /* Chain ordered by increasing
                                                        time */
        } /*End else */
} /*End while */
if(no family member running) { /* if #1 */
        Get head of queue of family members waiting to run;
        if(system entry time of first member on queue < = systime) { /* if #2
        */
                Put process on run queue;
        } /* End if #2 */
} /* End if #1 */
if (no currently running process) {
        Set flag to take process off run queue;
} /* End if */
if(systime mod 10 = 0) { /* 1/10 second has passed */
        Set flag to switch to new running process;
} /* End if */
if(systime mod 100 = 0) { /* 1 second has passed */
        Recompute priorities of all processes that have begun and not yet
        completed;
} /* End if */
if (there is a currently executing process) { /* if #1 */
        if(time for process to do I/O) { /* if #2 */
                Set I/O flag;
        } /*End if #2 */
        if(current process not member of family) { if #2 */
                if(time to exit) { /* if #3 */
                        Set exit flag;
                } /*End if #3 */
                if (time to do overhead operation) { /* if #3 */
                        Set overhead flag;
                } /*End if #3 */
        } /*End if #2 */
        else { /* current process is family member */
                if((current process has used all its CPU time) and (it has no
                        more I/O to do) and (it has no more paging to
                        do)) { /*if #2 */
                             Set exit flag;
                } /*end if #2 */
```

```
                    if(time to do paging) { /* if #2 */
                            Set paging flag;
                    } /* end if #2 */
            }  /* End else */
            if(time to adjust priority of current process) {  /* if #2 */
                    Adjust its priority;
            }  /* End if #2 */
    }  /* End if #1 */
/* End - clock algorithm */
```

Clockr, the clock routine used in the rescheduling experiment, followed the above scheme very closely. It only added two extra lines of code to the basic clock function that allow the special processes that are set up for this experiment to exit after the last family member had exited. To allow this to happen the clock routine had to be able to recognize special processes. Therefore, each special process was given a negative process ID while all other processes in the system had non-negative process ID's. When the last family member exited, a flag, endfam, was set. The additional code to allow the special processes to exit was as follows in pseudo-code:

```
if ((the process ID of the current process < 0) and
    (the endfam flag has been set){
            Set the exit flag; /* Allows process to exit */
}
```

Clockq was the clock routine used in the experiment examining the effects of assigning a definite quantum to a process about to get the CPU. Here too, the required change to the regular clock routine was minor. The test to determine whether the one-tenth second forced switch of the currently running process was due to occur on this tick was replaced by a test to check if the current process had used up its quantum. That is, the new test for replacing the current process in pseudo-code is:

if (quantum decremented by 1 = 0).

6.5.3 Overhead and I/O Routines

Both the overhead and the I/O routines were called by the main function on an iteration of the loop when the appropriate flag had been set. The I/O routine, doio,

handled both regular I/O and page faults and, as such, was called when either the flag for paging or the flag for I/O had been set. The algorithm here was rather simple. The number of blocks to be transferred was determined. For each such block, the time for its transfer from disk to main memory was computed using a randomly determined factor and was then added to the total time for the I/O operation. After the time to do the I/O operation had been calculated, the process blocked for that amount of time. The blocking of the process was simulated by having the process wait on the events chain for the time it took to do the I/O operation. The flag to switch the CPU to a new process was then set, and the function returned to the caller. The actual algorithm is shown below.

```
doio:

    if (no current process) {   /* Just to make sure */
            Return;
    {  /* End if */
    if(pgflg set) {  /* paging flag */
            Reset pgflg;
            numblks = 1;  /* Only one block for paging operation */
            Decrease number of pages yet to be paged in for this process by 1;
            Set number of ticks until next paging operation;
    }  /* End if */
    if (ioflg set) {  /* Flag for I/O operation */
            Reset ioflg;
            Increase number of blocks to be transferred by number of blocks per I/O
            operation for this process;
            Set number of ticks until next I/O operation;
            Decrease number of I/O blocks to be transferred for this process by the
            number being transferred in this operation;
    }  /* End if */
    Find minimum power of 2 greater than maximum value of transfer time;
    Use power of 2 found to create mask for random numbers to be generated;
    count = number of blocks to be transferred;
    while (count > 0) {  /* while #1 - for each block set transfer time */
            while(masked off random number is too large or too small) { /* while
                                                                        #2 */
                    Continue looping in this loop;
            }  /* End while #2 */
            iotime = iotime + time to transfer this I/O block;
            count = count - 1*
    }  /* End while #1 */
    System reentry time = systime + iotime; /* Time process will reenter queue */
    Insert process on events chain;
    current process pointer = NULL;
    Set flag to give CPU to new process;
/* End of doio */
```

The algorithm for the overhead routine, doohd, was very similar to that of doio. doohd first determined how long the process will be suspended and when the next overhead operation would take place. It then inserted the process on the events chain and set the flag to request the CPU be given another process. The pseudo-code follows.

doohd:

```
    Reset overhead flag;
    if(no current process) {
          Return;
    } /* End if */
    rmin = minimum possible overhead in simulation;
    rmax = maximum possible overhead at this point in simulation + number of
    processes in system;
    while (randomly determined overhead time not between rmin and rmax) {
          Continue looping; /* Try new random value */
    } /* End while */
    system reentry time = systime + overhead time;
    Set interval until next overhead operation equal to value of overhead variable;
    Actual interval till next overhead operaton = interval found - (number of
    processes in system)/10;
    Number of ticks to next overhead operation = absolute value of actual
    interval;
    Insert process on events chain;
    Current process pointer = NULL;
    Set flag to give CPU to new process;
/*End doohd */
```

The value of the "overhead" variable mentioned above depended on which family member was running. If no family member was running, "overhead" was set to a predefined constant.

6.5.4 Process Switching and Priority Calculation

The swtch function placed the currently running process on the run queue, dequeued the first process with the highest priority presently found on the run queue, and made this process the currently running process. It simply used the function insqpri to insert the current process on the run queue according to its priority and then called the appropriate dqrq function to get a new current process from the run queue. The dqrq function used in most cases just dequeued the process at the head of the run queue and

returned a pointer to this process. For the rescheduling experiment, the dqrq function, which is named dqrqr, also calculated the amount of time the process had just spent on the run queue waiting to run and incremented by one the number of times this process had been on the run queue. The quantum experiment also had its own dqrq function called dqrqq. In addition to the normal dqrq code, this version calculated the quantum to be given to the new current process as is described in the discussion of the quantum experiment later in this paper.

The function setprit was called by the clock routine to determine the new priority of the current process using the formula found in the real system. The real system, as has been mentioned previously, takes memory usage into consideration in computing priority. Since it was impossible to get statistics about memory usage at very close intervals in the lives of the processes in the real system, this factor had to be ignored.

The resched function was run every second of simulated time in the simulation to recompute the priorities of every process in the system. It used the formula for recomputing the priority found in the real system. This function first proceeded down the events chain, starting at the head, recomputing the priority of every process that was already running. It then recomputed the priority of the current process. Finally, new priorities were calculated for and assigned to every process on the run queue. This action was a bit more complex than it would seem since the order of the processes on the run queue was determined by their priorities. Therefore, each process on the run queue was dequeued, its new priority was calculated and set, and it was added to a temporary queue. When the new priorities of all the processes on the run queue had been found, they were all put back on the run queue. The function calpri did the actual computations. Using the formula from the real system, it computed the new CPU usage of a process. It then called setprit to determine the new priority and returned.

6.5.5 Exiting of Processes and Gathering of Statistics

The exiting of a process was handled by the function pexit which was called by main when the flag "exflg" was set by the clock routine. The pseudo-code for pexit follows.

```
pexit:

    Reset exflg;
    if (current process is last member of family){ /* if #1 */
            if (not run of the rescheduling experiment){ /* if #2 */
                    Set endsim; /* Flag to end simulation loop */
            } /* End if #2 */
            else { /* If is run of rescheduling experiment */
                    Set endfam; /* Flag that family has completed and
                            now special processes must end*/
            }
    } /* End if #1 */
    if (current process is special process for rescheduling experiment) { /* if #1 */
            Decrease number of special processes in system by 1;
            if (number of special processes < = 0) { /* if #2 */
                    Set endsim; /* All special processes have completed */
            } /* End if #2 */
    } /* End if #1/
    if ((rescheduling run) and ((current process is special process)
            or (current process is member of family))) {
                    Get statistics on time it spent on the run queue;
    } /* End if */
    Decrease number of processes in system by one;
    Pointer to current process = NULL;
/*End pexit */
```

When all family members and all test processes, if any, had completed, the simulation stopped, and main called prstats to print the elapsed time statistics for all family members. These statistics show how long it actually took for each member of the family to complete from the time it first entered the run queue in real simulated time. If this was a run of the rescheduling experiment, main then called expstats to print statistics about time spent on the run queue for all family members and special processes.

7. Debugging and Tuning the Model

7.1 Debugging

The first part of the debugging plan was to run the process initialization functions alone to ascertain that the output produced was what was expected. For these functions a main function was set up to call them and to deposit the values for each proc structure created in a human readable file, which is in a format that could be easily read and understood. For each function, the values generated for all the processes would be examined to make sure they were reasonable by comparing them to the ranges found in the data files. The first problem encountered was that the conditions of the loops that were placed in bpinit and apinit to make sure that reasonable values were obtained for the elapsed time in bpinit and for the number of disk I/O blocks in both functions at times caused infinite looping. Therefore, limits were added to the code so that after a certain number of tries, values are assigned to these variables rather than being obtained randomly and the loop is exited. After these modifications were made, the functions were run and the output was examined. This procedure was repeated several times using different seeds for the random number generator. No major problems were found with any part of the output produced by this test.

The next step was to test the queuing and dequeuing routines. A short function was written to create and initialize proc structures. The main function written for this test called this function and then used the queuing functions from the model to enqueue the proc structures on a queue ordered on system entry time, dequeue them, and then enqueue them on a queue ordered on priority. Each proc structure was printed out when it was created, and all proc structures on each queue were printed out to make sure that the queues were ordered properly and that all processes were accounted for. No major problems were encountered with the queuing routines.

The next phase of the testing and debugging plan was to run a small "simulation" and have the system print out appropriate fields of the processes on the queues in the order they are found on the queues. The purpose of this part was to test the clock function and the priority modifying functions. This test was first run with just three processes and later with about ten processes. The functions examined worked well and, therefore, no major changes were required.

The last phase of testing involved running the entire model first with a small set of processes and later on with a full set consisting of processes of all three types mentioned in the description of the model. Print statements were placed in all major routines. The events chain and run queue were printed out at definite intervals as was the proc structure of the current process. During these runs all the output from these print statements was collected in a file. The purpose of this phase of the testing was to make sure that the I/O routine was functioning properly, and that all of the parts of the program were interacting as intended. The output file produced was studied carefully to try to trace the progress of some of the processes. Running with a full set of processes was done many times each time seeding the random number generator with a different value and thereby getting a different process set each time. The one major problem found during these runs was that at times there were attempts to reference fields of the current process when in fact there was no current process. This oversight was removed by testing throughout the code for a current process before referencing its fields.

7.2 Tuning

7.2.1 The Goal of Tuning

It should be noted that the goal of the simulation was not necessarily to get a model that behaves exactly like the real system. The performance of the model should be sufficiently close to that of the real system in order that knowledge of the working of the

scheduler should be gained and experiments can be performed on the model. The model should be able to predict approximately what the impact of the modifications that were done in an experiment would be on the real system.

7.2.2 Overhead and Delay in the Real System

Three major areas of overhead and delay in the system were considered. The first area is the delay encountered when doing disk I/O. The overhead here is the time it takes to get a system buffer to put the data in and, when doing a read operation, the time waiting for the controller to retrieve the data. The other two causes of overhead and delay in the system were terminal I/O and housekeeping activities performed by the operating system. In UNIX, there does not appear to be a way of obtaining accurate statistics on how much time was consumed by either of these last two factors on a per-process basis. Furthermore, even general statistics on the amount of terminal I/O in the real system were not obtained. While this was admittedly an oversight, it is questionable how helpful such statistics would be. The amount of time spent by processes doing terminal I/O would still have to be estimated.

In doing output to a terminal, the process places the characters to be output on a queue and returns. Thus the only wait that could occur is in the case where the queue was full. Such a delay would be quite short under most normal circumstances. However, a process might have to wait significantly longer for input from the terminal. The delay would be firstly because of the time the user requires to think and type in the data. A second cause of delay, which in terms of time would be much more significant, is caused by the decision of the user not to continue interacting with the program for a while. In an operating system, where system delay is usually measured in hundredths or perhaps tenths of a second, this second cause of delay, which is often several seconds or even several minutes, could be very significant. Since no member of the family of processes did any terminal I/O, only the background processes would be affected by such delays.

No information at all can be obtained from the system as to how much overhead operating system housekeeping places on the system. The operating system itself keeps no statistics on this. However, the amount of time consumed performing such services as paging and carrying out scheduled operating system events, for example, the recomputation of process priorities, would probably be quite significant.

7.2.3 Method Used to Tune Model and the Results

All overhead was lumped together in the model. For each background process, two variables in the proc structure were used to control the overhead. The first variable specified the interval between overhead delays for this process. The second specified when the next overhead operation would take place for this process. A third variable collected the total time the process was suspended for overhead processing. This information was collected for potential statistical requirements. The method of dealing with overhead in the model was to suspend a process for a randomly determined amount of time which would represent some overhead that forced the process to wait. Only processes that were not family members had overhead associated with them. This arrangement was felt to be reasonable since family members are not interactive and waiting for terminal input is probably one of the more significant, if not the most significant, causes of overhead as defined in this project. Furthermore, a second major cause of overhead, paging, was estimated and applied to family members but not to other processes. The reason for arranging the model in this way was to allow a certain amount of control over the running of the model so that it would be easier to tune. Finally, it was felt that any overhead placed on family members by the operating system could be simulated by manipulating the overhead associated with the background processes.

The first step in the tuning process was to have the model generate an appropriately sized population one hundred eleven times corresponding to the one hundred eleven runs of the real system to which they were to be compared. The size of each

generated set falls within the limits for a set of processes for this project. For each run the random number generator was seeded with the next member of the fibonacci sequence starting 1,2,3,5,8... When the seeds became very large, only the lower twenty-four bits were used. The results in Table 1 show the average of the average elapsed times of the processes and other derived statistical quantities for this statistic (i.e., the average elapsed time) in both the model and the real system. While the averages and standard deviations of the average elapsed times are not that close, they are in the same order of magnitude and the ratios of the standard deviation to the mean for each system are close in value. These results are about as best as can be expected. As mentioned above, the statistic here is the average elapsed time in each run. Taking the average of such a statistic would tend to allow the most common range of values to have the most influence on the results and lessen the effect of uncommon values. However, the elapsed times in the model were generated randomly, and more or fewer extreme values could have been generated than were found in a run of the real system. Thus the most common range of values in a particular simulation run could easily be a bit different from that found in any recorded run of the family in the real system. Furthermore, the model did not take into account other external factors which would definitely have an impact on the statistics generated by the real system such as the number of users on the system while the family of processes ran. Thus the elapsed time of a given process in the real system is not a truly random event but is affected by exogenous factors about which no information has been obtained. Dr. Mitchell Small of the Department of Civil Engineering at Carnegie-Mellon University, whose work involves quite a bit of statistical analysis, studied the statistics obtained from the real system and the model. He feels that the model does roughly represent the real system and is as good a representation as can be expected under the circumstances.

Table 1
Values for Average Elapsed Time
(in seconds)

System	No. of Samples	Mean	Median	Standard Deviation	Standard Error of the Mean
Real System	111	998.3	744.2	792.8	75.2
Simulation	111	498.5	400.8	338.2	32.1

The second step of the tuning process was to adjust the values associated with I/O and overhead to obtain results for the members of the family of processes which fall within the range of values for each family member in the real system. This part involved running the model and gathering statistics on the actual elapsed times of family members in the simulation. The testing done in this step was actually done both before and after that of the first step of the tuning process mentioned previously. The reason such testing was done at the intial stage of tuning was merely to get a better idea of how well the simulation runs compared to the real system. This preliminary testing is, however, very significant in that early on it was observed that using CPU times to control the length of stay of a background process in the system produced results that were much farther from the results of the real system than use of elapsed time did. However, the part of the initialization routines that generated CPU times were retained because CPU time is used in the determination of the number of I/O blocks and because later use of the model may require it.

Those runs of the simulation that were done for this step after the general comparison of the system to the real system was made involved a great deal of work. The goal was to obtain empirically the best values for those constants that control the amount of time returned by the model for the duration of an I/O or overhead operation. The best results obtained was that all members were reasonably close to the bounds found in the real system in only 40% of the runs.

8. Experimentation

8.1 Areas Examined

This part of the project addresses first the issue of the fairness of the UNIX 4.2BSD scheduler to non-compute-bound processes and then attempts to determine if certain modifications can improve the performance of the scheduler.

The issue of fairness towards I/O-bound processes was first examined. This experiment investigated whether compute-bound processes get more CPU time than they should over a period of time and thereby unfairly delay other processes. To this end the experiment examined whether the formula used to recalculate the priority of all processes in the system every second boosts the priority of compute-bound processes, which are in the lower priority queues, too much.

The other experiments involved the effects of modifications to the scheduler. The first such experiment involved a specific part of the scheduling algorithm. As described in the proposal, every tenth of a second the scheduler places the currently running process on the run queue and allows the first process on the highest priority queue to run. The problem with this scheme is that a process requiring large amounts of CPU time may have to wait a long time to get the CPU and once it is given the CPU, may have to give it up after only one or two clock ticks. Thus the execution of the process is hampered by the scheduler. A fairer scheme would be to allow each process to be guaranteed a maximum time slice each time it gets the CPU and to let lower priority processes get larger guaranteed time slices. The assumption here is that since a process on a low priority queue has consumed a large amount of CPU time in the recent past, its CPU requirements will remain high. Such a scheme of increasing the time slice of a process as its priority drops is normally found in a multi-level feedback scheduler. The quantum experiment examined the effect of replacing the 4.2BSD mechanism with such a variable quantum scheme.

Finally, two experiments were conducted to observe what effect, if any, certain minor changes to the scheduler would have. The point of these experiments was to determine if some minor tuning could be done that would improve response time. The first of these experiments involved changing the number of queues in the scheduler. Although 4.2BSD provides one hundred twenty-eight different priorities, the number of

queues used by the scheduler on a VAX-11/750 is thirty-two. This number was apparently chosen so that the system could take advantage of certain VAX instructions to speed up the search for the highest priority non-empty queue by the scheduler when it must give the CPU to a new process. This experiment examined the effect of changing the number of queues to one, eight, sixteen, sixty-four, and one hundred twenty-eight.

The last experiment performed on the model was to observe the effect of delaying the reduction of the priority of a process. The priority of a process is reduced every time it consumes four ticks of CPU time. Two different sets of experimental runs were made in which the time between priority reductions was changed to eight and sixteen ticks respectively.

8.2 Explanation of the Statistical Analysis

Guidance in the area of statistical analysis in this part of the project was again provided by Dr. Small.

Since the method of statistical analysis of the data that was performed in each experiment was the same, a description of it is provided in the following paragraphs. All experiments involved fifty simulation runs each with a different seed of the random number generator. The elapsed time for the execution of each member of the family of processes was recorded in all experiments. In addition, in the rescheduling experiment described later, the time in clock ticks that each process of interest waited on the run queue for the CPU was recorded as was the number of times each such process was on the run queue. From these data, the average time spent on the run queue by each of these process could then be found. This statistic was of primary interest in that experiment.

For each statistic of interest, the mean, the standard deviation, and the standard error of the mean were calculated. The 90% confidence interval was then computed using the formula:

90% confidence interval = mean $\pm$ 1.645 * (standard error of the mean).

The meaning of the 90% confidence interval of the mean is that one is 90% confident that the true mean of the population under consideration is in that interval. In discussing the results with Dr. Small, it was learned that a significant change in a statistic between two populations is indicated if the two corresponding confidence intervals of the mean do not overlap.

The 90% confidence interval of the mean is used throughout this part of the project to check for changes in elapsed times caused either by modifications made to the model or by entering new types of processes into the model. It was also used in the rescheduling experiment to examine the statistics obtained from the model indicating the average length of time each process spent on the run queue. In all cases, it is assumed that a significant change has occurred only if the 90% confidence intervals of the mean found in the experimental run and its corresponding control run do not overlap.

Charts were also made for each family member in each experiment except for the rescheduling experiment showing the interval about the mean that is one standard deviation in length on each side of the mean. The purpose here is to see if any change of the model resulted in a smaller variance about the mean. Such a finding would indicate that a modification has helped make response times more uniform for that process. The degree of this improvement and the number of members of the family of processes so affected are brought into consideration in weighing its significance.

8.3 The Experiments

8.3.1 The Rescheduling Experiment

8.3.1.1 Design and Goals

This experiment was performed to determine whether a very compute-bound process would receive more than its fair share of the CPU at the expense of other

processes that are either I/O-bound or neither very CPU nor I/O intensive. The experiment required some changes to the clock function and to the function dqrq, which dequeues the first process on the highest priority "queue" and makes it the currently running process. These changes are described below.

The experiment consisted of two parts - a set of control runs and the runs of the experiment. In each part a group of special processes was run in addition to the background processes and the family of processes. The group of special processes consisted of one I/O-bound process and two "ordinary" processes in the control runs and one I/O-bound process, one compute-bound process, and one "ordinary" process in the runs of the experiment. The difference between the special processes and the others is that the former start at the beginning of the simulation run and continue to run until all members of the family complete. In addition to compiling statistics on the elapsed time of every family member, the amount of time that each special process and family member spent on the run queue and the number of times it was on the run queue were recorded. From these statistics the average amount of time each of these processes spent on the run queue each time it was on the run queue was determined. To obtain these statistics, the function dqrqt, which removes a process from the run queue, was replaced by the function dqrqr, which, as it removed a process from the run queue, recorded the amount of time the process had just spent on the run queue and increased the counter of the number of times this process was on the run queue by one. The function insqpri, which placed a process on the run queue, recorded the current time in the variable rqstart in the proc structure of the process to enable calculation of the time spent on the run queue. The clock function was replaced by a new function, clockr, as described in the section on the general design of the model that allowed the special processes to exit normally only after all the family members had completed.

8.3.1.2 Statistical Analysis of the Results

The charts in Appendix B labelled "rel" to "re7" show the 90% confidence intervals of the mean elapsed time for each member of the family of processes. The results as shown on these charts indicate that the introduction of a highly compute-bound process into the system does not have any significant effect on the members of the family of processes. There is no significant difference between the confidence intervals for the control runs and those for the experiment runs even though a highly compute-bound process in the experiment runs replaced one ordinary special process found in the control runs. As was pointed out in the proposal, one of the considerations for choosing members of the family of processes was that they consume a relatively large amount of CPU time. Thus if a compute-bound process did slow down the execution of other processes, some significant change would most likely show up in these results. However, in each case, the 90% confidence intervals of the mean for these processes overlapped greatly in comparing the control runs to the experiment runs. In fact, with the exception of "egrep", the results for the commands are almost identical in the control runs and the experiment runs.

The chart labelled "rql" shows the 90% confidence interval of the mean for the average time the process spent waiting on the run queue for the CPU for the compute-bound process in the experiment runs compared to the ordinary process corresponding to it in the control runs. In this case, the compute-bound process clearly had to wait longer to get the CPU than did the ordinary process since their respective 90% confidence intervals do not overlap. Since the UNIX 4.2BSD scheduler was designed to allow processes that consume less CPU time to get the CPU more often as is stated in the proposal, this result would be expected if the scheduler were working as intended because the compute-bound process uses significantly more CPU time than does an ordinary process. However, the other processes of interest did not have to wait significantly longer for CPU service in the experiment runs than in the control runs as can be seen in charts "rq2" to "rql0".

8.3.1.3 Conclusions of the Experiment

These results indicate that introducing a very compute-bound process into the system does not adversely affect the performance of the other processes. It appears that the scheduler is able to maintain stability with one such process in the system. It, of course, is not clear whether this conclusion would hold true if there were several highly compute-bound processes in the system at the same time. Thus the results of this experiment indicate that the scheduler is fair to other processes in the presence of a compute-bound process, but more experiments must be performed to verify this conclusion. In particular, it would be worthwhile to examine the case in which several compute-bound processes are present in the system simultaneously.

8.3.2 The Changing Quantum Experiment

8.3.2.1 Design and Goals

This experiment involved replacing the mechanism in the scheduler that forced a switch of the running process every tenth of a second regardless of how long the formerly running process had had the CPU. The new mechanism that replaced this scheme calculates a quantum of CPU time for a process just before it gets the CPU and allows it to keep the CPU for at most that quantum of time. The quantum assigned to a process is inversely proportional to its priority - the lower the priority the larger the quantum.

This experiment required changes to both the clock and the run queue dequeuing functions. The clock function had to be changed in order to allow the currently running process to be placed back on the run queue when its quantum expired and to eliminate the switching of the running process every tenth of a second that is found in the model of the real system. The function dqrqq is a modification of and replacement for dqrqt, which allows for the quantum of the process that is about to get the CPU to be calculated. This value is then placed in the global variable "quantum". The formula used to determine the quantum is:

quantum = QUANTUM + qfactor * (priority/qdiv-12).

where: QUANTUM = 10

qfactor is a variable the value of which is specified before the run of the model by the user and is constant for the run of the model.

priority is the current priority of the process.

qdiv = 4

Dividing the priority by four specifies the number of the queue in which the process currently belongs. Since the model considered only priorities of fifty and above, the highest priority queue in the model is queue number twelve. The formula gives any process in that queue a quantum of ten. Lower priority processes get larger quantums depending on their priorities and the value of qfactor for this run.

Five sets of fifty runs each were made in this experiment. For each set of runs, qfactor was set to a different value. The values used were one, two, four, eight, and sixteen.

The goal of this experiment was to compare the fifty runs of the unmodified model to each of these five sets of experimental runs to observe whether this new mechanism would cause any significant change in response time. To this end the same fifty seeds for the random number generator were used in the runs of the unmodified model and in each set of the experimental runs. The 90% confidence intervals of the mean elapsed time for each member of the family of processes in all five sets of runs were compared to those obtained from the set of runs of the unmodified model. Similarly, the interval one standard deviation in width on each side of the mean was determined for each family member for all sets of runs.

8.3.2.2 Statistical Analysis of the Results

Charts seq1 to seq7 show the 90% confidence intervals for each member of the family of processes in each set of runs. The interval labelled "Actual" shows the confidence interval of the unmodified model. Charts sdq1 to sdq7 show the intervals of

width one standard deviation on each side of the mean. "Actual" on these charts also refers to the unmodified model.

Although the results for the 90% confidence intervals for the family in the experiments show no definite improvement over those of the unmodified model, there seems to be a slight improvement in most family members as the number of queues increase. The only exceptions to this observation are the "egrep" and "w" commands. However, it would appear that modifying the model in the ways done in this experiment has little effect on performance. The charts showing the interval of one standard deviation on each side of the mean also do not indicate any real improvement over the unmodified system. This result is really surprising since this modification guarantees that a process that has just gotten the CPU will receive a definite amount of CPU time. Thus the problem of processes getting very little CPU time once they are allowed to run would be eliminated. It was also thought that the mechanism in the real system resulted in rather large variances in the elapsed times for processes, and the modifications done in this experiment would result in smaller variances being observed.

8.3.2.3 Conclusions of the Experiment

It was felt that the part of the scheduler that switches the running process every tenth of a second sacrifices a certain amount of responsiveness and uniformity of response time for simplicity. Although this assumption may be true, modifying the scheduler to assign a quantum to a process based on its CPU usage did not cause definite improvement in either area. Although there seemed to be some improvement in mean response time, this result is indefinite since the 90% confidence intervals of the experiments still overlap the confidence intervals derived for the unmodified model. Therefore, it seems clear that the modifications done in this experiment as described above will not by themselves improve the response time of the system. Furthermore, as noted in the previous section,

these modifications will not decrease the variance of the mean response time for the processes studied. These results indicate that alterations to the scheduler such as those done in this experiment may prove useful but only in conjunction with other modifications to the scheduler.

8.3.3 Changing the Number of Queues

8.3.3.1 Design and Goals

The purpose of this experiment was to observe the effect of changing the number of priority queues on the system. Although the scheduler allows for one hundred twenty-eight priorities, there are only thirty-two queues. Therefore, to find the proper queue for a process, the operating system divides its priority by four. The model was modified to simulate different numbers of queues by varying the number by which the priority is divided to obtain the number of the queue on which to place the process. If, for example, the priority is divided by eight, the result will be sixteen queues in the model. Dividing the priority by two will cause the model to simulate a system with sixty-four queues. Aside from allowing this division to vary among the different sets of runs, no other modifications to the basic model were required for this experiment.

It was not assumed that having a different number of queues would improve the response times of the members of the family of processes. This experiment is just an investigation of what effect, if any, such a change would have. It may be argued that having fewer queues would benefit compute-bound jobs at the expense of I/O-bound jobs because, in such a case, the former would be more often placed in higher priority queues and compete with the latter for the CPU in a round-robin fashion. If they are in a lower priority queue, they might have to wait until all of the higher priority processes completed or blocked before they could get the CPU. On the other hand, it could be that the rescheduling mechanism works well enough that the more compute-bound processes do

not remain on lower priority queues for long periods without executing. Thus this conjecture might not really be valid, and changing the number of queues in the system might not affect compute-bound processes very much at all.

The experiment consisted of five sets of fifty runs each in which each set of runs had a particular number of queues. For the five sets of runs the numbers of queues were one, eight, sixteen, sixty-four, and one hundred twenty-eight.

8.3.3.2 Statistical Analysis of the Results

As in the changing quantum experiment, a chart for each member of the family of processes was drawn containing the 90% confidence interval of the mean for each set of runs of the experiment and for the set of runs of the unmodified model. The charts are labelled secql to secq7.

It appears from the charts that the only version of the model that showed significant improvement over the unmodified model is the one with only one queue. This particular version is, of course, simply a round-robin style scheduler and was made part of this experiment simply to see how it would compare to the multi-level feedback scheduler as implemented in 4.2BSD. It was to be expected that the members of the family of processes would fare better in this scheme since they do use relatively large amounts of CPU time compared to other commonly run commands. In a multi-level feedback scheme, these processes are probably often on lower priority queues while the more I/O-bound processes are on the higher priority queues. Therefore, they must wait until all the higher priority processes complete or block or the rescheduling scheme boosts their priorities. But in a round-robin scheme, all processes are on one queue and compete for the CPU equally. As such, compute-bound processes would not be delayed by higher priority processes. Of course, in such a scheme, interactive processes would not be as responsive as they would be in a multi-level feedback scheme. Thus, in an interactive environment such as is found in UNIX, a round-robin scheme would be undesirable.

Charts sdcq1 to sdcq7 contain the interval of one standard deviation on each side of the mean for each member of the family of processes. Each chart contains the intervals for each set of runs of the experiment and the set of runs of the unmodified model for each member of the family of processes. It is clear from the graphs that changing the number of queues does not reduce the variance unless only one queue is present.

8.3.3.3 Conclusions of the Experiment

It would appear that changing the number of queues has no significant effect on the model except for the case of one queue. Neither the 90% confidence interval nor the variance appears to change very much at all, and certainly not enough to be considered significant. Therefore, it seems that thirty-two queues is no better nor worse a choice for the scheduler than any other possibility that was tried in this experiment. In fact, it makes a good deal of sense to have thirty-two queues on a VAX-11 machine as opposed to a larger number since the system can take advantage of some of the special instructions that it would not be able to use with a larger number of queues. On the other hand, while fewer queues could be used with the fast VAX-11 instructions, such a change would probably slow the response of interactive processes in general for the following reason. The priorities of the compute-bound processes would decrease as in the real system; but whereas with thirty-two queues, it takes four drops in priority to place a given process on the next lower queue since the priority is divided by four to get the queue number, in a system with sixteen queues, the priority would have to decrease by eight before the process was placed on the next lower queue. In the meantime this compute-bound process would remain in the higher queue and compete for the CPU with the interactive processes for a longer period of time. Allowing fewer queues would not be very desirable according to this assessment. Therefore, the regular system with thirty-two queues appears to be the best approach under the circumstances.

8.3.4 The Delay Priority Drop Experiment

8.3.4.1 Design and Goals

The 4.2BSD scheduler decreases the priority of a running process every fourth clock tick that it consumes. The purpose of this experiment is to observe the effect on response time of delaying the priority reduction. Two sets of fifty runs each were run. For the first set the priority was reduced every eighth clock tick consumed; in the second set, the reduction was done every sixteenth tick consumed. No modifications to the basic model were required for this experiment because when this experiment is chosen, the user is prompted for the number of ticks between priority reductions. This value is entered directly into the model.

It was thought that by postponing the reduction in priority the response time may decrease. The hypothesis was that this change would allow a process to remain in a higher queue longer and thereby allow it to get the CPU more quickly rather than being delayed in a lower queue waiting for higher priority processes to exit or block or for an increase in its priority that would put it into a higher priority queue.

8.3.4.2 Statistical Analysis of the Results

As in the previous experiments, the elapsed times of all members of the family of processes were recorded for each run in each set of runs. From these data the 90% confidence interval of the mean elapsed time for each member was computed. Additionally, the interval one standard deviation on each side of the mean for each family member was computed. Charts ses1 to ses7 show the 90% confidence intervals, and charts sds1 to sds7 contain the standard deviation intervals. Each chart shows the plots for a given family member for the unmodified model, the version in which the priority was decreased every eight ticks, and the version with the priority decrease occurring every sixteen ticks.

It is clear from the 90% confidence intervals that slowing the priority change of processes has little, if any, effect on the elapsed times of the family members. It appears to make little difference whether the drop in priority is done every four, eight, or sixteen clock ticks consumed by the process. The charts showing the standard deviation intervals also consistently show no major change in the variance for any member of the family of processes when the dropping of priority is slowed down.

8.3.4.3 Conclusions of the Experiment

It is quite clear that increasing the number of clock ticks before the priority is dropped has no effect on the response time of any member of the family of processes. It is unclear why such a change should have no effect on the scheduler. It could be that if this modification were combined with a modification to the formula used to compute the priority such that each time the priority drops it does not decrease as much as it does presently, some improvement would occur.

9. Conclusions of the Project

9.1 Summary of the Project and Its Results

The purpose of this project as described in the introduction was to examine processor scheduling in UNIX 4.2BSD by means of a computer simulation of the scheduling system upon which various experiments were performed. The experiments were planned to be a first attempt to study the quality of the process scheduling system by examining the performance of one part of the scheduler, the rescheduling algorithm, and by testing modifications to the scheduler that attempt to improve the response time of the operating system.

The effort to create a model of the scheduler produced a reasonable representation of the real system. The model did not perform as close to the real system as was hoped,

but it was sufficiently close. Some factors that were unknown and difficult to estimate, such as the number of users on the system at a given time, contributed to the inaccuracy of the model.

9.2 Conclusions Drawn

The experiments appear to demonstrate two basic points. First, the rescheduling scheme that raises the priorities of all processes every second does not seem to favor compute-bound processes. The second point is that the scheduler appears to resist efforts to tune it. Any attempt to improve its performance would probably require quite a bit of redesigning.

The first point is apparent from the results of the rescheduling experiment. Compute-bound processs do not appear to monopolize the CPU or even hamper the progress of other processes.

The second point is a general conclusion based on the experience gained from the other experiments. The modifications to the scheduler were just simple changes to its makeup. The fact that none of the changes significantly affected the response times of the members of the family of processes would indicate that the scheduling system requires more complex modifications to influence response times.

One final point should made. These experiments provide us with no clue as to how effective the scheduler is. While, on the one hand, no better performance was gained by the modifications, on the other hand, none of the modifications degraded the performance of the system. It would appear that the only way to evaluate the effectiveness of the scheduler would be to design a rather different one and compare its performance to that of the present scheduler. A computer simulation of the proposed design could be performed to determine whether it would in fact be an improvement before embarking on the relatively difficult task of the detailed design and coding of a new scheduler.

9.3 Shortcomings of the Experiments

There are two major areas in which the model appears to be lacking. The first involves the difficulty encountered when tuning the model. The trial-and-error method of tuning the model by adjusting the variables associated with I/O and overhead did not bring the behavior of the model as close as desired to that of the real system. While tuning the model in this way did improve its behavior in that the average elapsed times of the members of the family of processes were roughly similar to those found on the real system, these statistics should have been much closer. It would probably have helped if statistics on terminal I/O had been gathered. However, given the complexity of the real system, it is unclear how well the model of the scheduler could be tuned even with such statistics.

Some information about the number of users logged in taken at frequent intervals would also have been useful. With such information, a history of the system could have been constructed for each run. Such histories would have given a more detailed view of system activity over all the runs and could have been used to constrain the generation and progress of processes in such a way that would more closely reflect the real system.

The second shortcoming of the project is the fact that no I/O-bound processes were made part of the family of processes. One of the requirements set down for a command to become a member of the family of processes was that it consume a relatively large amount of CPU time. The reason for this requirement was to ensure that the command be useful for tuning the model. Although I/O-bound processes would not have aided the tuning process, they would have been useful in the experiments to see how the changes in the model affected their performance. It would be expected that an improvement in the performance of the compute-bound members of the family is at the expense of I/O-bound processes. However, the question is how adversely the I/O-bound processes are affected. If the performance of the I/O-bound processes degrades significantly, the modification

made might not be very useful. The ideal result would be some gain in the performance of compute-bound processes with little impact on I/O-bound processes.

A general limitation of the approach taken in this system is the fact that the scheduler is just part of a large and complex system the parts of which work together. Overall performance of a process is a result of how well these parts work both independently and jointly. A change in one component that appears to improve general performance in the model might not have much effect when added to the real system simply because the change in the scheduler adversely affected other parts of the system. For example, if every time a process is given the CPU it must execute a page fault, overall performance will suffer no matter how much better the new scheduling scheme might appear to be. There was a conscious attempt in this project to minimize such problems by making only modest modifications to the scheduler. However, the effect of any change cannot be known until the modified scheduler is used on the real system.

9.4 Lessons Learned

9.4.1 Improvements for the System

One of the most obvious and important areas of improvement for the system is the creation and handling of overhead. It would have been helpful to run a process named "overhead" that would not be scheduled but would take over the CPU at various times to simulate system housekeeping. This process would be used in addition to the overhead scheme currently in place in the model. This addition would make tuning somewhat easier.

A major problem with the system is that the data were obtained only at a load level of two even though the daemon was constructed to fire up the family of processes at higher load levels. The system upon which the daemon was run, however, rarely reaches a load level of four. Thus the data used as input to the model only represents a moderately

loaded VAX-11/750. As such the project is limited to simulating just such a system. An improvement would be to gather data on a system that often ran at higher load levels. The problem of slow response times on a VAX-11/750 is more noticeable at load levels greater than two. The weakness of the scheduler would most likely be more obvious under higher load levels since its effectiveness in maintaining good response times would probably deteriorate as the load level increased. Without having run the experiments under such conditions, it is difficult to predict what results they would yield if the load level were raised significantly. Therefore, simulating a UNIX 4.2BSD system on which the load level is often fairly high would be quite valuable.

A minor modification that would improve ease of use of the simulator would be to allow the user to specify the number of runs he wants. As presently constructed, the program performs only one execution of the simulation when started. However, any serious use of the model requires many runs in order to generate a sufficient amount of data for evaluation. To accomplish this in the project, all experiments were run by means of a shell script. While this method caused little trouble, the proposed change would allow the program to execute the runs faster, and it would eliminate the need for a shell script that must know certain details specific to the simulator such as the names of the output files.

Another minor improvement would be to allow a non-interactive mode. As is currently implemented, the program prompts the user for necessary information. The program should allow this information to be specified on the command line.

9.4.2 Future Experiments

In all of the experiments one factor was allowed to vary while the others were held constant. Since none of these changes alone had an effect on the scheduler, it would be reasonable to try various combinations of these modifications to observe whether they would affect performance. Such an experiment would consist of making one type of

change and holding it constant while varying a second change. For example, one might want to observe the effect of both initiating a quantum based system and changing the number of queues. While he would set the quantum to be determined by a certain fixed formula, he would try different numbers of queues. He could then perform experiments holding the number of queues constant while using different formulas to determine the quantum.

Another experiment that could be attempted would be to modify the experiment in which the change of priority of processes was slowed down. Instead of calculating the priority from the time consumed, only a fraction of the time would be used. For example, if the priority reduction is slowed to be reduced every eight ticks, one might want to calculate the priority using half the time consumed. In such a scheme, not only does the process remain in a higher priority queue longer, but also when it does change queues, it is placed in a higher queue than that in which it would otherwise be put. Thus the process should definitely receive better CPU service.

9.4.3 Future Thesis Topics

The two major versions on UNIX in general use are AT&T's System V and Berkeley UNIX. The scheduler modelled in this project is the one found in Berkeley 4.2BSD. Berkeley has recently released 4.3BSD which contains some small changes in processor scheduling. Each of these versions of UNIX contains variations of what is basically the same general scheduling scheme. One potential area of research for a future thesis would be a computer simulation which compares all three of these scheduling schemes. Such an investigation would study how these schemes perform relative to each other.

Another area of investigation is how well the scheduler in 4.2BSD responds to increasing the load level of the system. The original intention of this project was to create a model of the scheduling system using runs of the real system at different load levels. Unfortunately, it became difficult to obtain the necessary statistics from the

proposed target system, and, therefore, the project had to be changed to one in which a simulation was done of a moderately loaded system and which involved a certain amount of experimentation. Such a project would require that the model be tuned such that its performance is reasonably close to the real system at all load levels.

A third area of interest would be a model of a system with a system resources monitor (SRM) running on top of the scheduler. An SRM controls the length of the run queue and the progress of each process in the system by estimating how much of the system resources each process on the run queue should have consumed at a given point in its life and suspending those processes that have consumed more than that amount. The suspended processes remain in that state until their resource consumption is considered acceptable for their age. Such a higher level scheduling system is found in IBM's MVS system and can be purchased for the IBM VM system. The project would concentrate on large non-interactive processes and try to determine whether an SRM would have any overall positive effect on UNIX system performance.

10. Key Words and Phrases

background processes - those processes run in the simulation merely to simulate the system load.

CPU time - amount of time a process has had use of the CPU.

current process - the process that is currently executing.

latency - time it takes for a disk to rotate so that desired data are adjacent to the read-write head.

process blocking - voluntary giving up of the CPU by a process.

process ID - number assigned to a process that uniquely identifies it to the system.

process suspension - the act of taking away the CPU from the current process by the system.

quantum - specific time interval for which a process has use of the CPU.

seek time - time it takes for a disk drive to move the read-write head from its current position to the cylinder containing the desired data.

APPENDICES

Appendix A: The Proposal

A Computer Simulation of Processor Scheduling in UNIX 4.2BSD

Michael D. Grossman

Table of Contents

1. Introduction and Background

1.1 Problem Statement

Every time-sharing operating system has some scheme for scheduling the CPU. This scheme chooses which runnable process should get the CPU next. The goal of this project consists of two parts. The first part is to model the processor scheduling scheme found in UNIX 4.2BSD either in the computer simulation language GPSS or in some higher level general purpose language and to tune the model until it yields response times comparable to the real system for a given family of processors. The second part of the project is to experiment with the model to determine whether the scheduling algorithm can be improved to yield better performance for various types of processes and a fairer scheduling scheme. The model will represent the real system being run on the MSBVAX machine in the Division of Biostatistics of the School of Medicine and Dentistry of the University of Rochester. This machine is a VAX-11/750.

1.2 Library Search Results

A library search was conducted primarily to look for similar work that was done by others. Several guides to general computing literature were consulted in addition to journals dealing exclusively with computer simulation. The guides to computing literature included "Computer and Control Abstracts" (January, 1983 to June, 1985), "Computer Literature Index" (1980 to 1982), the "ACM Guide to Computing Literature" (1981 to

1983), and the "ACM Computing Reviews" (1981 to 1985). The simulation journals that were searched are the SCS journal titled "Simulation" (1978 to 1984) and "Simuletter" (1976 to 1983) published by the ACM. Also, the proceedings of the Summer Computer Simulation Conference and the Winter Simulation Conference for the years 1981 to 1984 were examined for papers dealing with simulation of the UNIX 4.2BSD scheduler.

The goal of this search was to determine if any work similar to that proposed in this paper had been done and to find any material that might enhance or facilitate this project. Most of the investigations of UNIX performance involved the AT&T systems rather than the BSD systems. Furthermore, I did not find any articles or books describing a computer simulation of any UNIX operating system. The performance studies I found were conducted by running benchmarks. However, the information obtained in one of these papers may prove to be helpful. That paper is "Measuring and Improving the Performance of 4.2BSD" by Sam Leffler, Mike Karels, and M. Kirk McKusick (USENIX CONFERENCE PROCEEDINGS SUMMER 1984, June 12-14, 1984, Salt Lake City, Utah, pp. 237-252).

The paper describes the benchmarks run by the Berkeley team and others to examine the performance of 4.2BSD. The original version of the system was found to consume an unacceptable amount of CPU time, and subsequently changes were made to decrease system overhead. The authors include the statistics from the original version and from the improved

4.2BSD system. Although this paper is not very useful for the actual construction of a simulation model, the statistics given for system CPU time, system calls, and interrupts should be of value.

The two other papers that seemed relevant concerned themselves primarily with the AT&T versions of UNIX, "The Evolution of UNIX System Performance" by J. Feder (AT&T BELL LABORATORIES TECHNICAL JOURNAL, vol. 63, no.8, October, 1984) describes the results of various enhancements to the AT&T systems that were intended to improve its performance. Feder shows how these changes improved performance of succeeding versions of UNIX over the past eight years.

The paper "An Experimental Investigation of Scheduling Strategies for UNIX" by Darwin R. Peachey, Richard B. Bunt, Carey L. Williamson, and Tim B. Brecht (PERFORMANCE EVALUATION REVIEW, vol. 12, no.3, pp. 158-66, August, 1984) deals with the effect of certain changes to the CPU scheduler and the swap scheduler. Here too the information is not very relevant since they discuss only the AT&T systems.

1.3 Glossary

CPU - The central processing unit of a computer.

Family of processes - The group of processes which will be run by a daemon at specific load levels. They will be selected either to study their behavior or to gather statistics.

I/O - Abbreviation for input and output.

Load level - The total number of processes on all run queues at a given instant.

Nice - A command to run a process with a lower priority or, for the superuser, to either raise or lower the priority of a process. A user either specifies in the command a certain numerical value by which the priority will be changed or uses the default value. The term also refers to this numerical value.

Process ID - The unique identification number assigned to a process by the operating system.

Process scheduler - That part of the operating system which decides which process will get the CPU next.

Resident set size - The number of pages in main memory at a given time that are owned by a particular process.

Response time - For the purposes of this project, it is the amount of time a process requires to complete measured from the time at which it was first placed on the run queue.

1.4 Description of Real System and Modelling Goals.

The UNIX processor scheduling scheme is basically a multi-level feedback scheme. This scheme determines which process gets the CPU next primarily on the basis of the priority of the process. The system has several priority levels and to each level there corresponds one queue. Each process with a given priority that is waiting to run is placed on the queue for that priority. The next process to run is taken from the highest priority queue on which there are processes. If there

are several processes on that queue, they are removed in round-robin fashion from the queue and run until no more processes are waiting for the CPU at that priority level. At this point, the system will look for the next lower priority queue containing processes and run processes from that queue in the same manner as described above. This procedure is repeated down to the lowest level queue. The priority of a process is based on the amount of CPU time consumed. The priority of a process is inversely proportional to the amount of CPU time consumed - the more CPU time used, the lower the priority.

This is the basic scheme used by 4.2BSD UNIX. In 4.2BSD it is implemented in the following way. There are one hundred twenty-eight priorities. Lower numerical values represent higher priorities. On a VAX-11/750, the hardware provides thirty-two queues. 4.2BSD makes use of these queues for processor scheduling by dividing the priority calculated by the scheduler by four.

A process receives a priority in one of two ways. The first way is that a priority is assigned to it by the scheduler. A process can receive a priority in this way whether it is executing user or kernel code. The second way of receiving a priority can only occur in the kernel when the process is about to wait on an event. In this case, the code that is executing will assign it a predefined priority. After the event has occurred, the process will proceed with this priority.

The scheduler assigns priorities starting at fifty. Lower priorities are assigned only in the second manner described above.

Clock interrupts occur every one hundredth of a second. At every clock interrupt, the scheduler will compute a new lower priority for the currently running process if the process has consumed four clock ticks of CPU time since the last time its priority was lowered. If the process has been running at a priority assigned to it by the scheduler, the newly computed priority will become its actual priority. However, if the process has been running at a priority assigned to it by the kernel code that was executing as described above, its priority will not change. However, the computed priority will be stored and later assigned to it when it leaves the kernel.

The formula the scheduler uses to compute the new priority is:

Priority = Min(ticks,255)/4+PUSER+2*(nice-NZERO)

where:

ticks = amount of CPU time used by the process in clock ticks

PUSER = 50

nice = nice value of process

NZERO = 20

Min(x,y) = smaller of the two values x and y

In addition to this computation, if the amount of free memory in the system is less than the desired amount and the resident set size of the current process is greater than the maximum resident set size it should possess at this point, the priority is decreased by adding eight to the above calculation. If the priority found is greater than one hundred twenty-seven, the value is set to one hundred twenty-seven.

Every tenth of a second, the scheduler forces a switch in the running process. This enables the system to give processes of equal priority a chance to run. The purpose of this part of the scheduler is to allow processes in the highest priority queue in which there are processes to run in a round robin fashion.

Every second the priorities of all the processes in the system are recomputed so that the priority of processes that have not run in a while will increase. Here, as above, the computed priority becomes the priority of the given process only if its present priority was assigned to it by the scheduler. The formula used to recompute the priorities is:

Priority=Min(ticks,255)*loadav*2/(2*loadav+1)+nice-NZERO

where:

loadav=the one minute average of the number of processes on the run queue. This quantity is recomputed every five seconds.

ticks=amount of CPU time used by the process in clock ticks.

nice=nice value associated with the process.

NZERO=20

Min(x,y)=smaller of the two values x and y.

If a process uses more than ten minutes of CPU time it is given a "nice" value of four.

While the proposed model of this system might not contain structures and arrangements that look like the UNIX 4.2BSD scheduler, it should accurately model the function of the real system. To drive the model, actual system statistics will be obtained and fed into the model. Statistics will be taken at various load levels and will always be taken with a predefined family of processes running in addition to whatever processes are running at that time. The model will be calibrated to represent the functioning of this family of processes reasonably well under all test cases. That is, the response times of all members of the family produced by the model should compare reasonably well with those gathered from the real system.

2. Project Description

2.1 General Outline

The project I propose to do can be broken down into eight parts. I will list each step and later describe each one in detail.

1. Identify the processes that are appropriate for inclusion in the family of processes.
2. Create the family of processes.

3. Construct the daemon. The daemon will be designed to fire up the entire family of processes at pre-specified load levels.
4. Gather statistics.
5. Build the model.
6. Run and validate the model.
7. Experiment with the model.
8. Draw conclusions.

2.2 Indepth Description of Each Step

2.2.1 Identification of Potential Family Members

Those processes that are typical of the types of processes that often run on a general computing machine in an academic environment and use significant amounts of processor time will be considered for membership in the family. From this group of processes the members of the family will be selected. The latter requirement is necessary to ensure that the members of the family will be useful in calibrating the model so that it will yield response times comparable to those of the real system, and, therefore, will accurately represent the system. Processes that do a great deal of I/O compared to the amount of CPU time they consume are not good candidates for the family since the CPU scheduler is the system being modelled, and time spent doing I/O is considered an exogenous variable. Although I/O bound processes do affect system behavior, their performance would not

be significantly affected by the slight changes to the model necessary for tuning it since they consume relatively little CPU time.

2.2.2 Creation of the Family of Processes

The family will consist of a group of processes that typify the kinds of programs run on a general purpose machine in an academic environment and a group of processes that gather statistics from the system. It is expected that the family will include the following user programs:

1. Compilations of a Pascal program and a C program.
2. One shell script using the ex editor to create or modify a file.
3. Runs of the "w", "egrep", and "nroff" commands.
4 Statistics gathering programs that run the commands "ps axl" and "ps axv" at two second intervals and the "vmstat" command at five second intervals. All of the programs will write the results out to files.

The actual programs that make up the family of processes may differ from what is described here if the indications from preliminary research are that the amount of CPU time consumed by any of these types of programs is small. They are only mentioned here since they are typical of the categories of programs generally seen running on UNIX systems in an academic environment.

2.2.3 Construction of the Daemon

The daemon will be made to fire up the processes at system load levels of two, four, six, and eight. "Load level" here is taken to mean the average number of processes in the run queue over a period of one minute.

The daemon will examine the load level every five seconds. If the load level is between two to three, four to five, six to seven, or eight to nine, the family of processes will be fired up. The statistics gathering processes will be started first. The user processes will then be started after giving the statistics gathering processes enough time to get started. The statistics gathering processes will be allowed to run until all the user processes have completed. After the family has run, the daemon will sleep for five minutes before examining the load level again. In this way it is expected that the processes running every time the daemon fires up will be different. Thus each run should show the family executing under at least slightly different conditions.

2.2.4 Statistics Gathering

Statistics gathering is the last part of the preliminary work to be completed before building and running the model. The retrieval of all the necessary statistics consists of two parts. First, system accounting must be turned on. The accounting system will create a file containing entries for each process that terminates while accounting is turned on. A

program will be written to process this data so that all the pertinent information found in the system accounting files will be obtained for each process. The "sa" command normally used to process these files cannot be used since it only gives a summary of each command run and does not give information about individual processes in the form that is required by this project. The information that is found in system accounting that is relevant to the model is the amount of CPU time each process used, the total amount of time each process was in the system, the number of disk I/O blocks transferred to the process, and the time of day that the process started.

The second part of statistics gathering is the running of the three statistics gathering processes that are part of the family of processes and are running while the user processes run. Each of these processes will direct its output to a file. One process will run the command "ps axl" at two second intervals. Relevant to the project, this process will yield for each process in the system its process ID, its parent's process ID, its "nice" setting (process scheduling increment), its priority, and its resident set size. The second process will run the command "ps axv" at two second intervals. This command will furnish the number of pages read in for each process. The third process will run "vmstat" at five second intervals, to obtain the size of the memory free list. Although these commands yield much more information, it appears that this is all that is required for the project.

The results of the statistics gathering will be used to construct a profile of an average process that was running while the family of processes ran. This "average process" will be used to design an initial model. To create a model of the full system, the average process will be replicated such that the load on the system will be similar to that found whenever the family of processes ran. To simulate a real instance of the system, the resource usage of each of these clones will not be the average usage but will be allowed to vary randomly within the bounds found from the statistics for each resource. This group of processes plus the family of processes will be the input for the model.

It should be noted that while statistics will be gathered for all the processes that are in the system while members of the family of processes have not yet completed and all such processes will be represented in the model, the model will be calibrated only for those processes in the family. That is, while the model will approximate the behavior of the other processes that existed when the family had run, as was mentioned above, the model will not be calibrated to represent each of these processes. These processes merely provide the background load for the model that was present at any time the family of processes ran. However, the model should accurately represent each member of the family under all load levels and other conditions that are examined.

2.2.5 Building the Model

The model will be constructed to function as the processor scheduling system does in UNIX 4.2BSD.

In the initial design of the model, priorities will be set only by the scheduler. The mechanism of setting priorities within the kernel code that is executing rather than computing them based on CPU time will be ignored. It is assumed that the amount of time a process spends at these high priorities is relatively small and, therefore, disregarding this method of priority setting will not affect the ability of the model to accurately represent the real system. If this assumption appears to be false when the model is run, the function of setting these high priorities will be added to the model.

The following is a description of the proposed model assuming the model is written in GPSS. The CPU is a pre-emptable facility. Each process is a transaction carrying with it all the necessary statistics as parameters. The ready-to-run queue is a user chain ordered by process priority. Process priority is NOT the GPSS priority but rather the value assigned to the process by the model and stored in a parameter. In addition to its priority each process has parameters containing: 1) the process ID, 2) the amount of CPU time the process has left, 3) the number of I/O blocks yet to be read in or written out to disk, 4) its "nice" setting, and 5) its resident set size. All user processes run at GPSS priority 3. At system starting time several other transactions are created.

One runs in a loop splitting off user process transactions at the proper times. A second one runs in a loop and splits off a clock interrupt transaction every .01 second with GPSS priority 0. A third transaction runs in another loop and splits off a transaction to switch the CPU to another process of the same priority every .1 second. This transaction runs at priority 2. Finally, a fourth transaction splits off transactions every second to reset the priorities of all processes in the system. This transaction runs at priority 1.

When a user process seizes the CPU, it holds it until one of three events occurs. 1) It may block for I/O. A determination is made based on the statistics taken when a given process should block for I/O. A process transaction will have one parameter holding the amount of time it is to run the next time it gets the CPU. If it is scheduled to block for I/O, it will release the CPU and stay suspended for the amount of time it will take to complete the I/O. The amount of CPU time consumed up until it blocked will be deducted from the amount of CPU time left. 2) It is forced to give up the CPU by a clock interrupt at a .1 second interval as described above. The currently running process is enqueued on the ready-to-run queue at the end of the group of processes with the same priority. A new process from the highest priority queue at which there are processes, is chosen to run. This process is the first one on the queue at that priority level. 3) It encounters a clock interrupt at a one second interval as described above. In this

case, the priorities of all processes in the system are recomputed, and if any runnable process has a priority higher than that of the currently running process, that higher priority process gets the CPU.

When the amount of CPU time left for a user process reaches zero, the process exits. The model will run until all members of the family of processes have completed and exited.

2.2.6 Running the Model

The model will be run using the statistics as described above. The amount of simulated time it takes each family member in the simulation run to complete will be compared with the same statistic that was produced by the real system. When these statistics are sufficiently close for all members of the family of processes and for all runs of the simulation, then it will be assumed that the model yields acceptably good results for the family of processes. If there is a relatively large discrepancy between the system statistics and the results of the runs, the model will be adjusted. There are several variables which will have to be estimated and can be adjusted. These include the amount of time that was used by the system to perform certain housekeeping tasks, and, therefore, was not part of the accounting system time of any process, the actual amount of time to fulfill any I/O request, and the actual time at which a process blocked for any I/O request. While the second can be approximated by knowing the disk transfer rate and estimating

the delay due to contention, the first and third will be more difficult to estimate. It is expected that most if not all adjustments to the model will involve these variables. The model will be considered valid when the results of each run of the family of processes with each distinct group of other processes and its load level agree with the results produced by the real system at the given load level.

2.2.7 Experimentation

After the model is validated, the process scheduling algorithm will be modified and experimented with to see what effects such modifications have on response times. The 4.2BSD System Manual states that CPU scheduling in 4.2BSD "tends to favor interactive processes and processes that execute only for short periods."[1] The first experiment that is anticipated is to test this assertion by examining the effect of allowing either an extremely compute-bound process or an extremely I/O-bound process to run in the model. Other possible experiments that may be attempted are changing the number of queues, changing the number of time slices consumed before the priority is changed, and increasing the size of the time slice given to a particular process as its priority decreases.

2.2.8 Drawing Conclusions From the Model

The results of the experiments on the model will be studied carefully. An assessment will be made of the scheduler

[1]William Joy and others, 4.2BSD System Manual (Berkeley, CA, 1983), p.16

as it now exists and whether any of the functional modifications done to the model could improve its ability to meet the goal quoted above from the 4.2BSD System Manual.

3. Deliverable Items and Milestones

The items that will be produced and incorporated into the thesis are a listing of the model and a report showing the results of the runs of the model and how they compare with the statistics obtained from the real system. In addition, a detailed discussion of both the calibration and validation of the model and the further experiments that were carried out on the validated model will be included in the thesis.

The following are the milestones for this project.

1. Running the daemon and gathering statistics. This milestone will be met when we have gathered three days worth of statistics.
2. Designing and coding the model. This will be met when the coding for the model is completed and will compile correctly.
3. Validation of the model. Validation will be considered accomplished when the response times of the model are sufficiently close to the actual response times of the real system that were determined from the statistics.
4. Experimentation and the compilation and discussion of the results of the experiments. This milestone

will be met upon the completion of the report concerning these experiments that is included as part of the thesis.

5. The final milestone will be the completion of the discussion of the entire project and the conclusions drawn from it.

4. Qualifications

I work as a systems programmer on UNIX 4.2BSD systems. I have been a systems programmer for almost three years and have been working on UNIX systems for well over a year. My duties involve long term programming projects to tailor the operating systems to the needs of my employer. I am familiar with the structure of the UNIX 4.2BSD operating system and with some parts of the source code. In preparation for this project I have studied the relevant portions of the UNIX 4.2BSD kernel.

In addition I have taken Operating Systems I (ICSS 809), Operating Systems II (ICSS 810), and Modelling and Simulation I (ICSS 730) at R.I.T.

5. Bibliography

Feder, J. "The Evolution of UNIX System Performance", AT&T BELL LABORATORIES TECHNICAL JOURNAL, vol. 63, no. 8 (October, 1984), 1791-1814.

Joy, William, Eric Cooper, Robert Fabray, Samuel Leffler, Kirk McKusick, David Mosher. 4.2BSD System Manual. Computer Systems Research Group, Computer Science Division; Department of Electrical Engineering and Computer Science; University of California, Berkeley; Berkeley, California: 1983.

Leffler, Sam, Mike Karels, M. Kirk McKusick. "Measuring and Improving the Performance of 4.2BSD", USENIX PROCEEDINGS, Salt Lake City, Utah (June 12-14, 1984), 237-252.

Peachey, Darwyn R., Richard B. Burt, Carey L. Williamson, Tim B. Brecht. "An Experimental Investigation of Scheduling Strategies for UNIX", PERFORMANCE EVALUATION REVIEW, vol. 12, no. 3 (August, 1984), 158-166.

Appendix B: The Charts

Charts for the Rescheduling Experiment

Introduction

This set of charts illustrate the results of the rescheduling experiment. This experiment investigated whether a compute-bound process would receive more than its fair share of the CPU at the expense of other processes. A compute-bound process was created and run in addition to several other special processes consisting of an I/O-bound process and one or more "ordinary" processes. These ordinary processes are processes that look like processes that were created by the normal initialization routines. Each of these special processes began at the start of the simulation run and ended execution after all the family members had exited. Corresponding to each experimental run, a control run was made in which the compute-bound process was replaced by an ordinary special process.

Charts re1 to re7 show the 90% confidence interval of the mean elapsed time for each member of the family of processes in both the control and experiment runs as labelled on the charts. Charts rq1 to rq10 show the 90% confidence intervals for the average time the process spent waiting on the run queue for each special process and each family member in both the control and experiment runs.

Chart re1

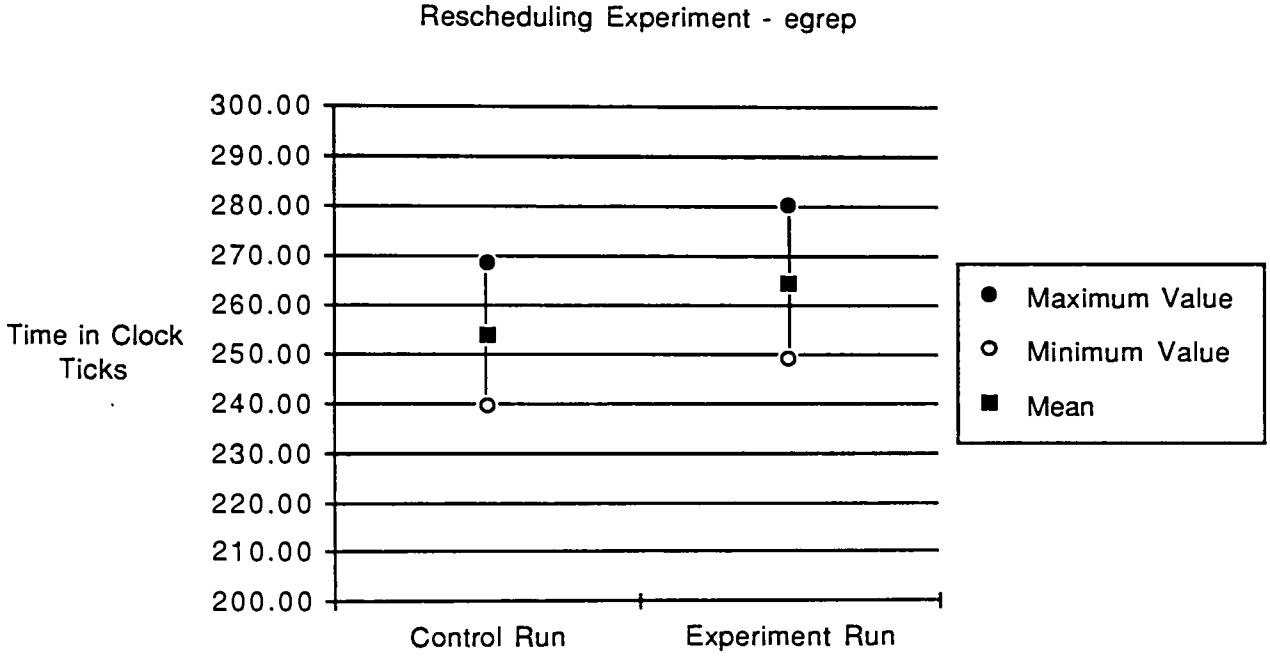

Rescheduling Experiment - egrep
Time in Clock Ticks
300.00
290.00
280.00
270.00
260.00
250.00
240.00
230.00
220.00
210.00
200.00
Control Run
Experiment Run
Maximum Value
Minimum Value
Mean

Chart re2

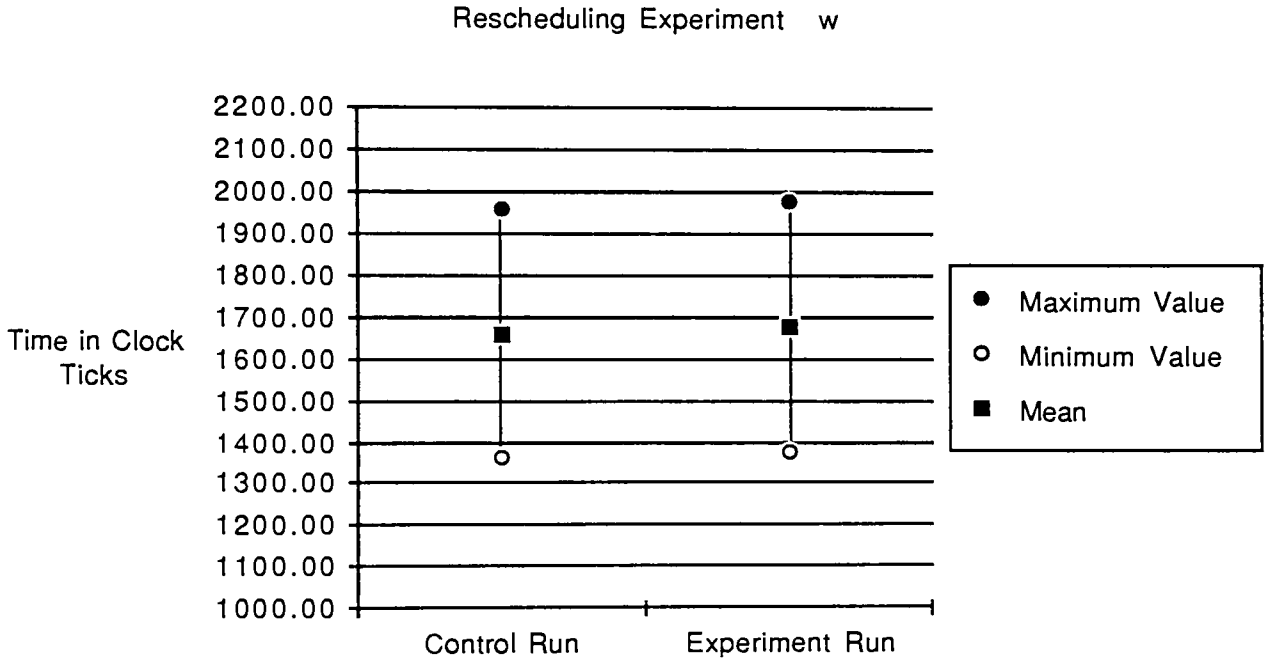

Rescheduling Experiment w
Time in Clock Ticks
2200.00
2100.00
2000.00
1900.00
1800.00
1700.00
1600.00
1500.00
1400.00
1300.00
1200.00
1100.00
1000.00
Control Run
Experiment Run
Maximum Value
Minimum Value
Mean

Chart re3

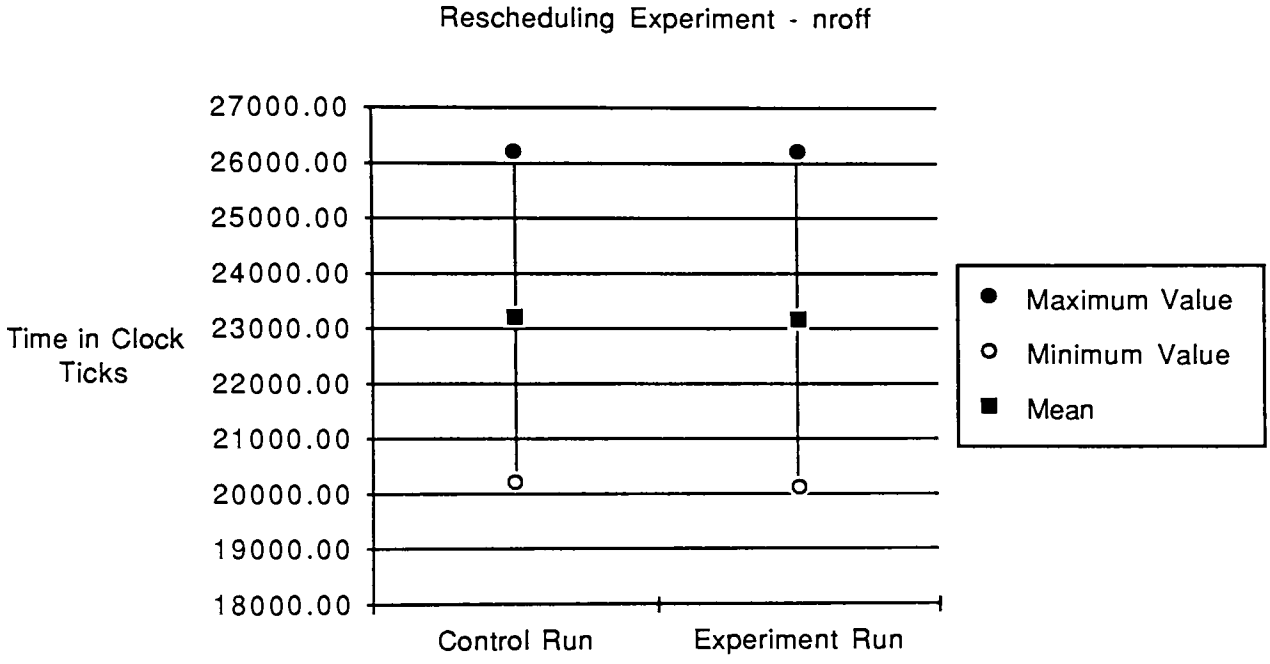

Rescheduling Experiment - nroff
Time in Clock Ticks
27000.00
26000.00
25000.00
24000.00
23000.00
22000.00
21000.00
20000.00
19000.00
18000.00
Control Run
Experiment Run
Maximum Value
Minimum Value
Mean

Chart re4

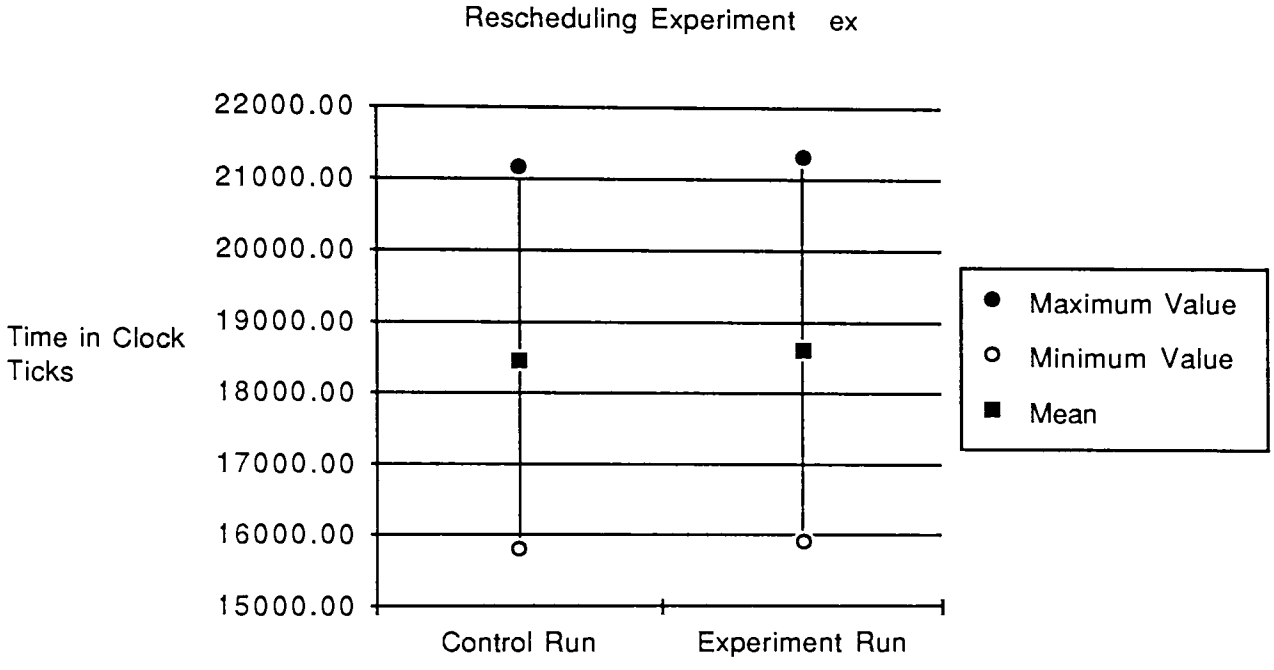

Rescheduling Experiment ex
Time in Clock Ticks
22000.00
21000.00
20000.00
19000.00
18000.00
17000.00
16000.00
15000.00
Control Run
Experiment Run
Maximum Value
Minimum Value
Mean

Chart re5

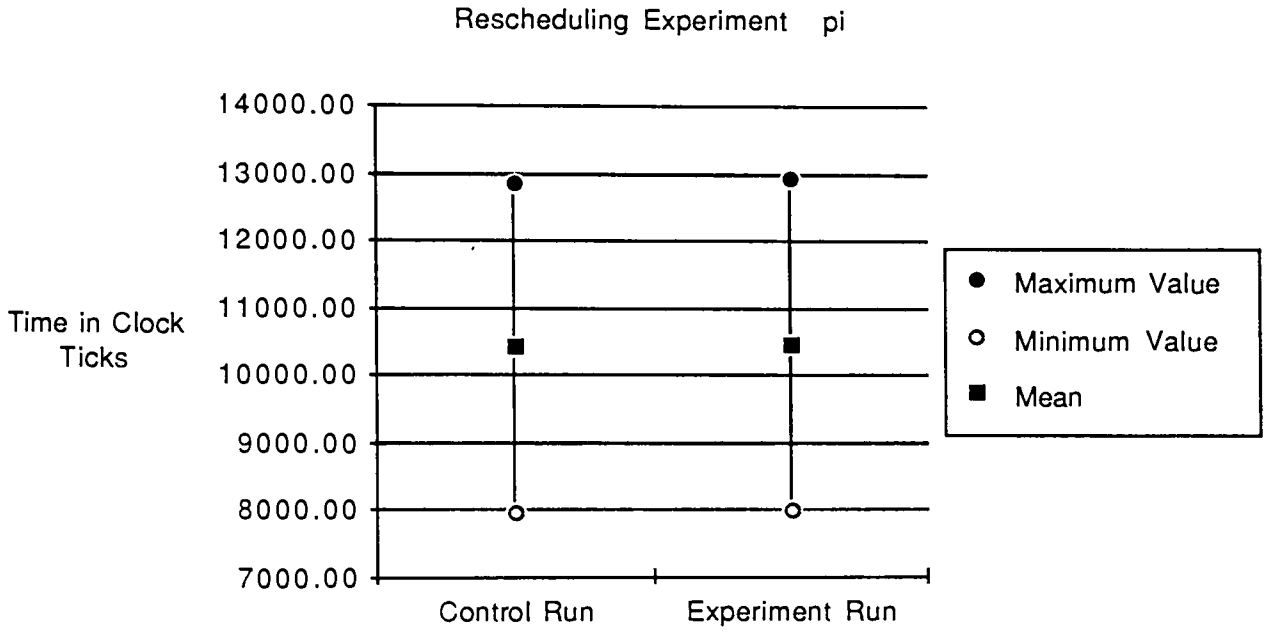

Rescheduling Experiment pi
Time in Clock Ticks
14000.00
13000.00
12000.00
11000.00
10000.00
9000.00
8000.00
7000.00
Control Run
Experiment Run
Maximum Value
Minimum Value
Mean

Chart re6

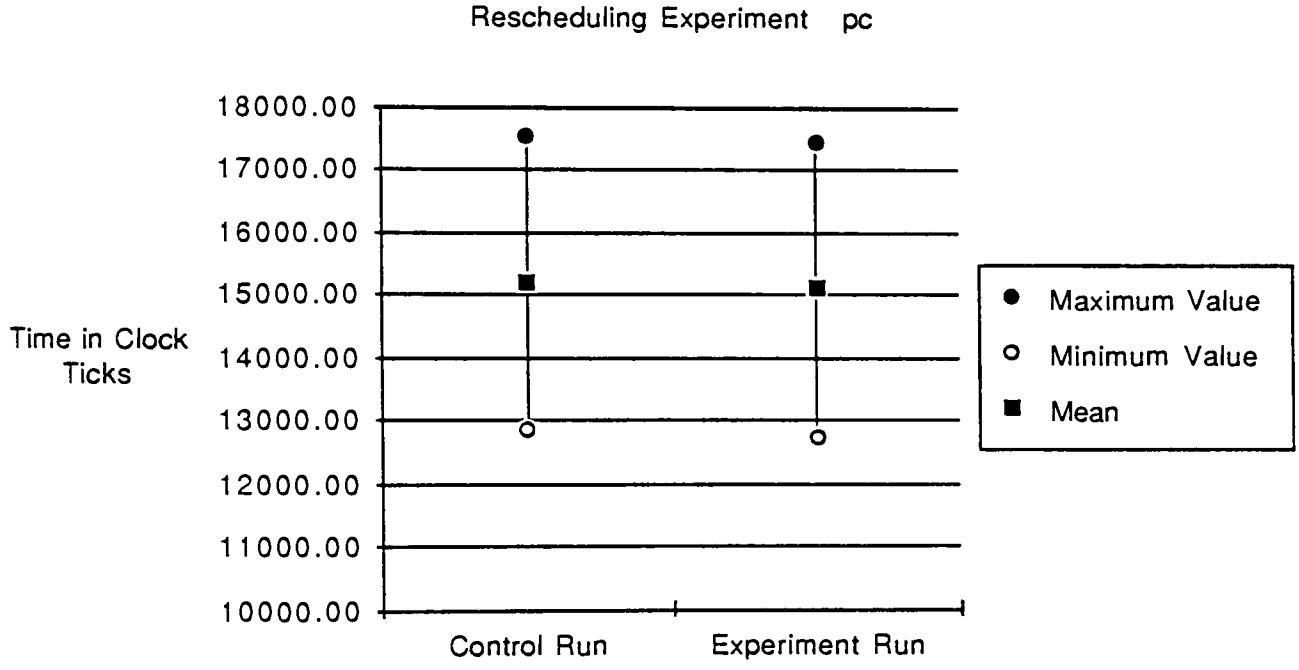

Rescheduling Experiment pc
Time in Clock Ticks
18000.00
17000.00
16000.00
15000.00
14000.00
13000.00
12000.00
11000.00
10000.00
Control Run
Experiment Run
Maximum Value
Minimum Value
Mean

Chart re7

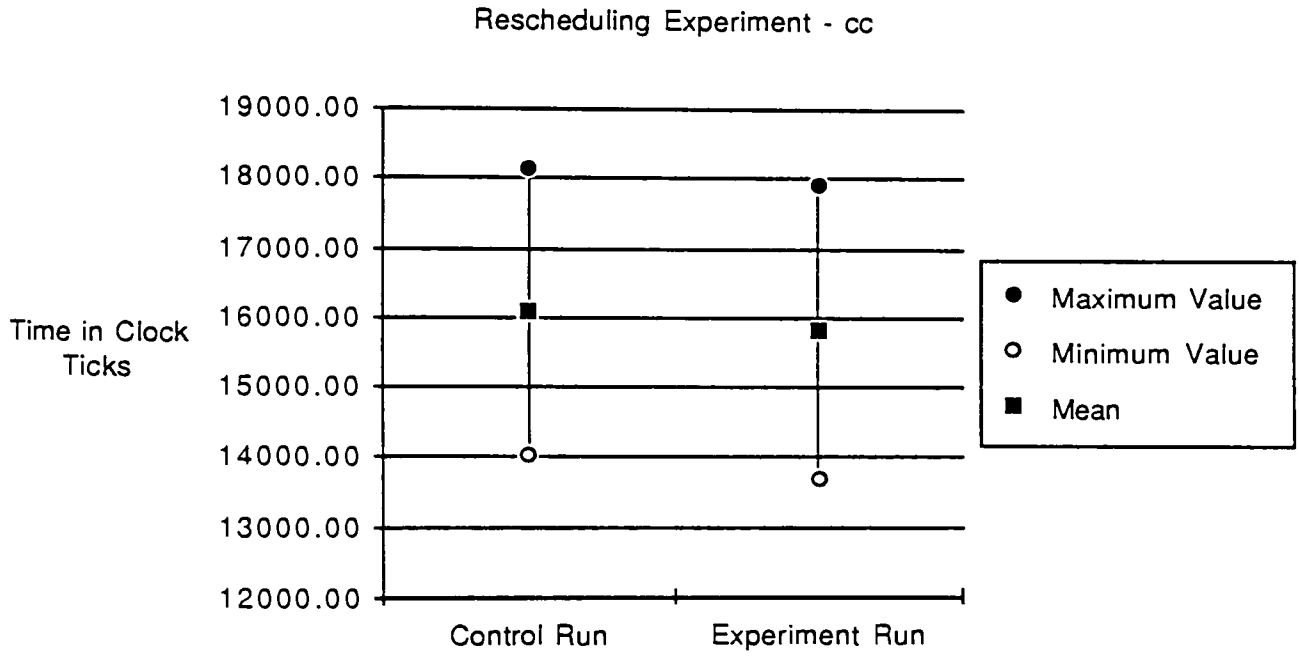

Rescheduling Experiment - cc
Time in Clock Ticks
19000.00
18000.00
17000.00
16000.00
15000.00
14000.00
13000.00
12000.00
Control Run
Experiment Run
Maximum Value
Minimum Value
Mean

Chart rq1

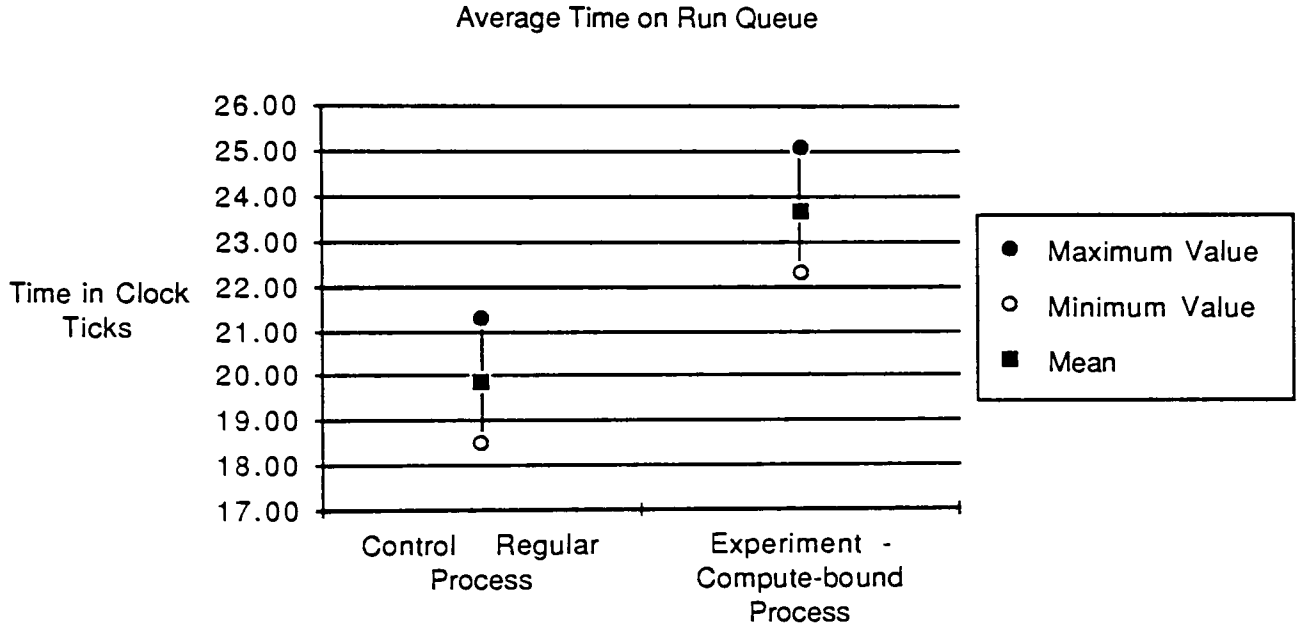

Average Time on Run Queue
Time in Clock Ticks
26.00
25.00
24.00
23.00
22.00
21.00
20.00
19.00
18.00
17.00
Control Regular Process
Experiment - Compute-bound Process
Maximum Value
Minimum Value
Mean

Chart rq2

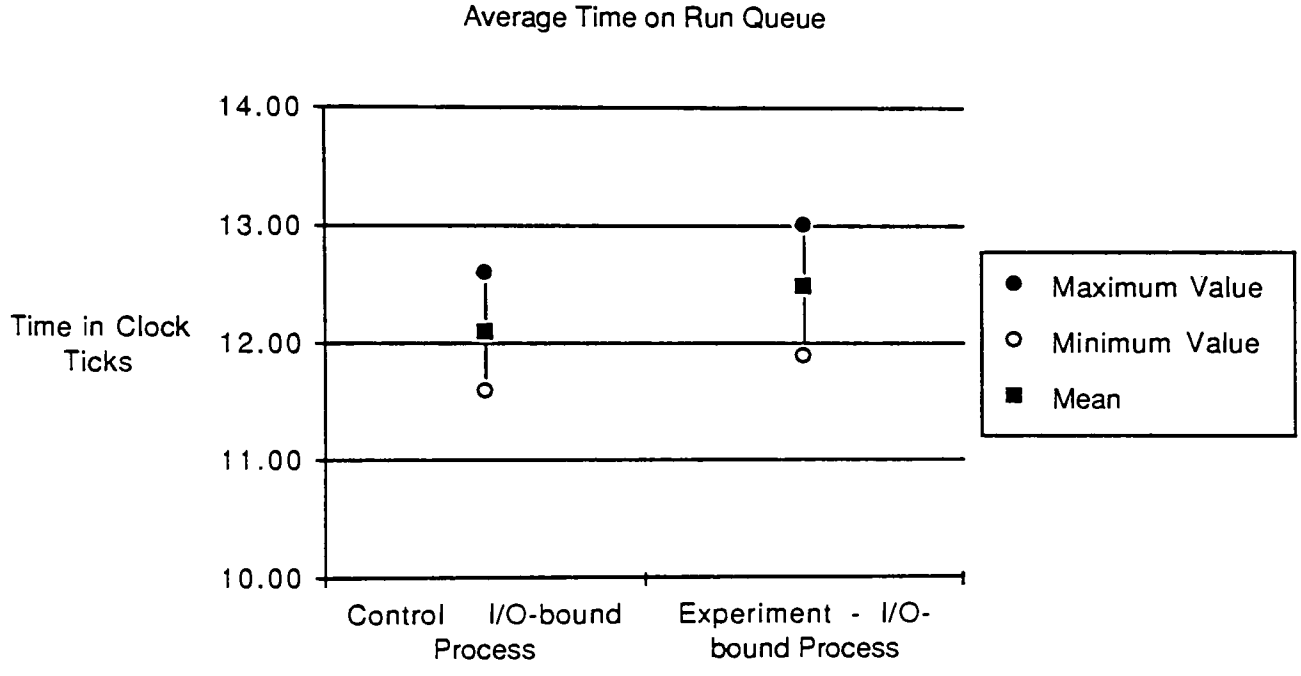

Average Time on Run Queue
Time in Clock Ticks
14.00
13.00
12.00
11.00
10.00
Control I/O-bound Process
Experiment - I/O-bound Process
Maximum Value
Minimum Value
Mean

Chart rq3

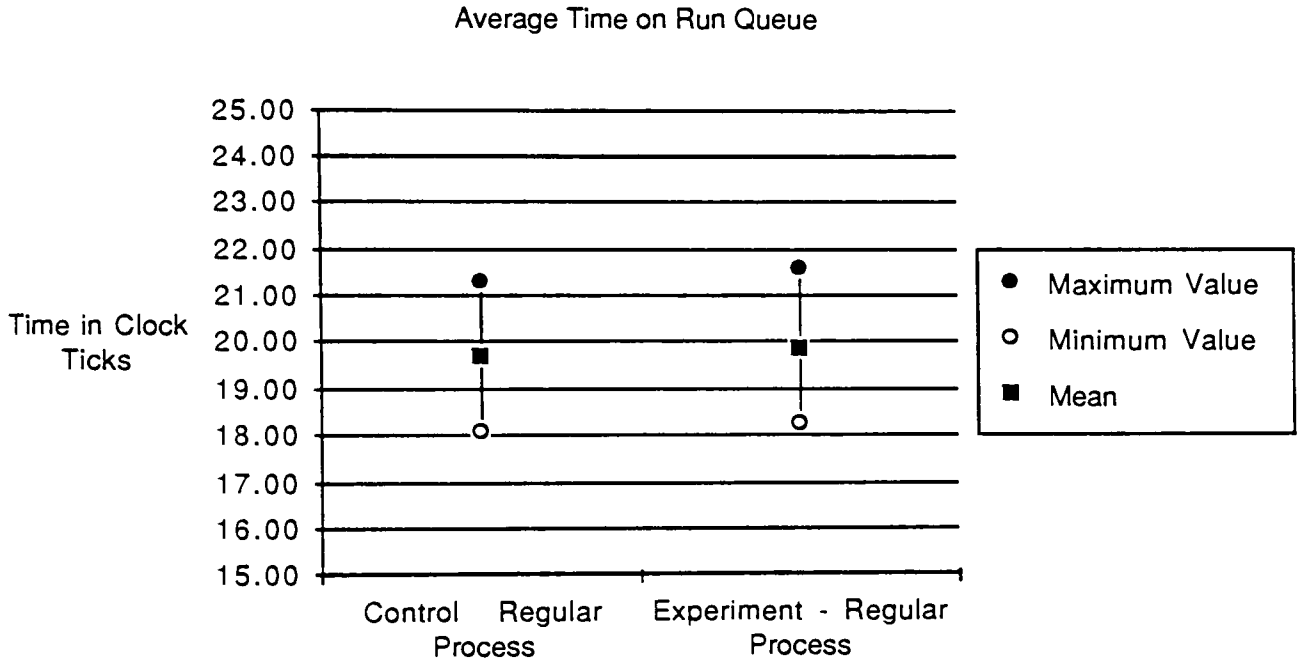

Average Time on Run Queue
Time in Clock Ticks
25.00
24.00
23.00
22.00
21.00
20.00
19.00
18.00
17.00
16.00
15.00
Control Regular Process
Experiment - Regular Process
Maximum Value
Minimum Value
Mean

Chart rq4

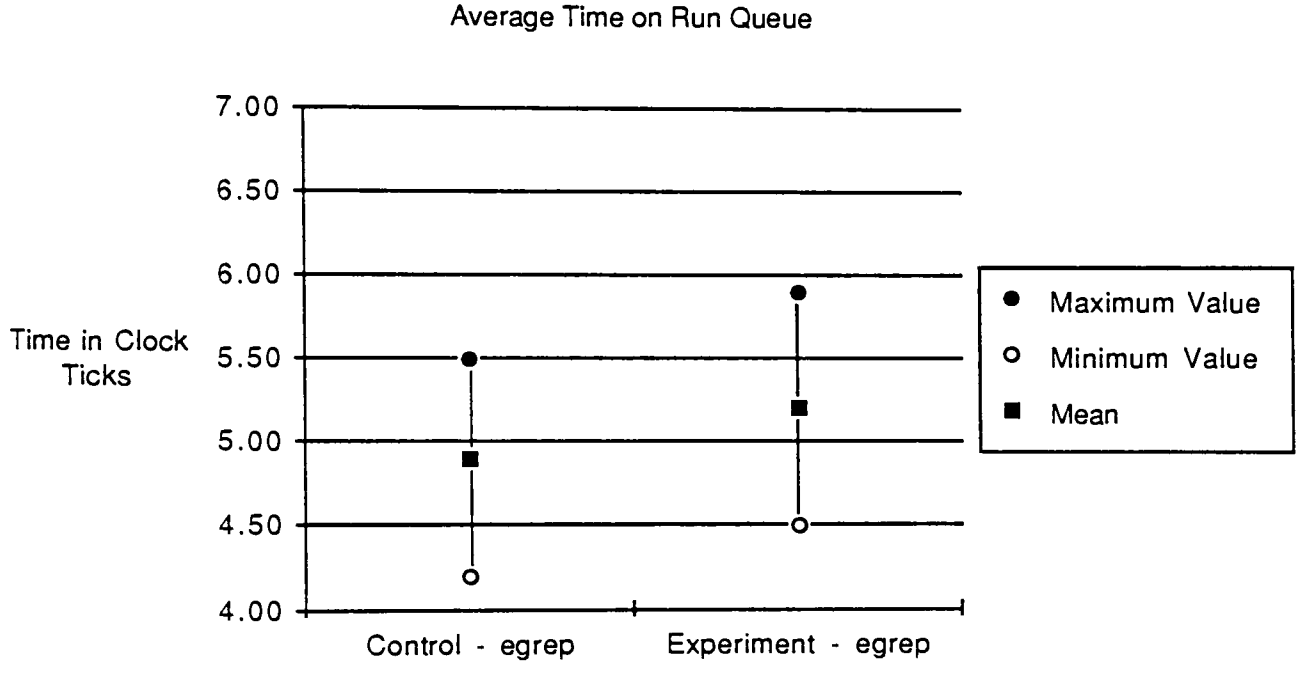

Average Time on Run Queue
Time in Clock Ticks
7.00
6.50
6.00
5.50
5.00
4.50
4.00
Control - egrep
Experiment - egrep
Maximum Value
Minimum Value
Mean

Chart rq5

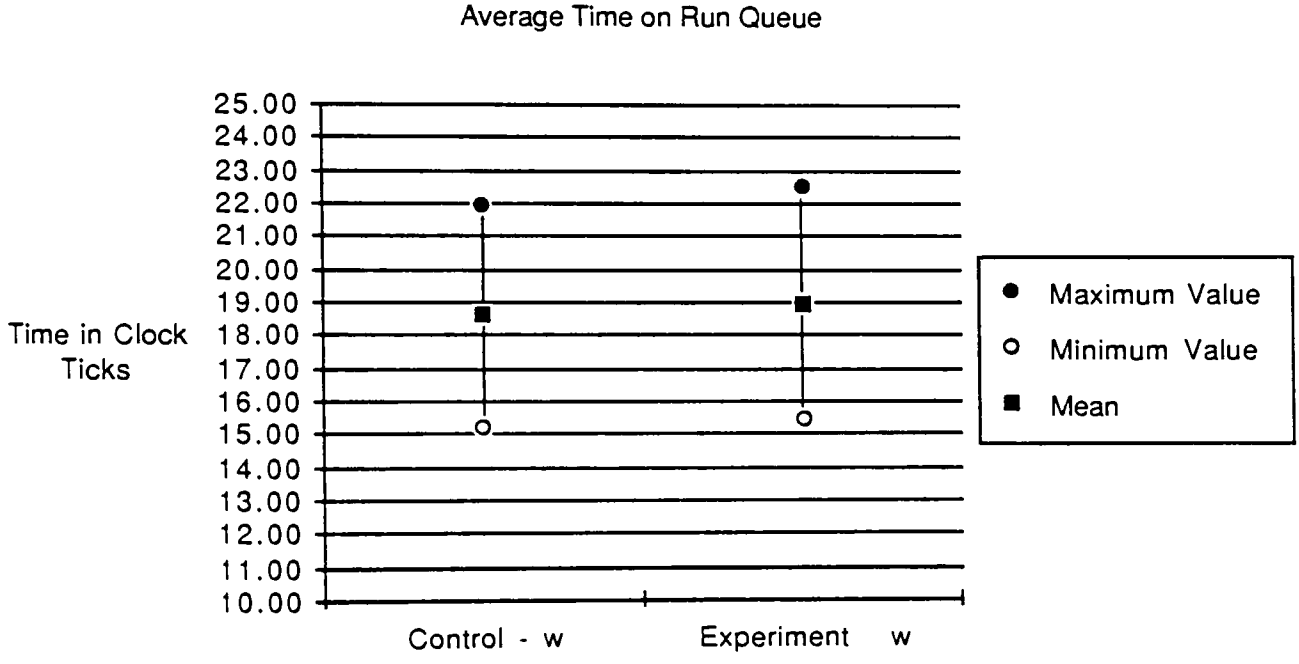

Average Time on Run Queue
Time in Clock Ticks
25.00
24.00
23.00
22.00
21.00
20.00
19.00
18.00
17.00
16.00
15.00
14.00
13.00
12.00
11.00
10.00
Control - w
Experiment w
Maximum Value
Minimum Value
Mean

Chart rq6

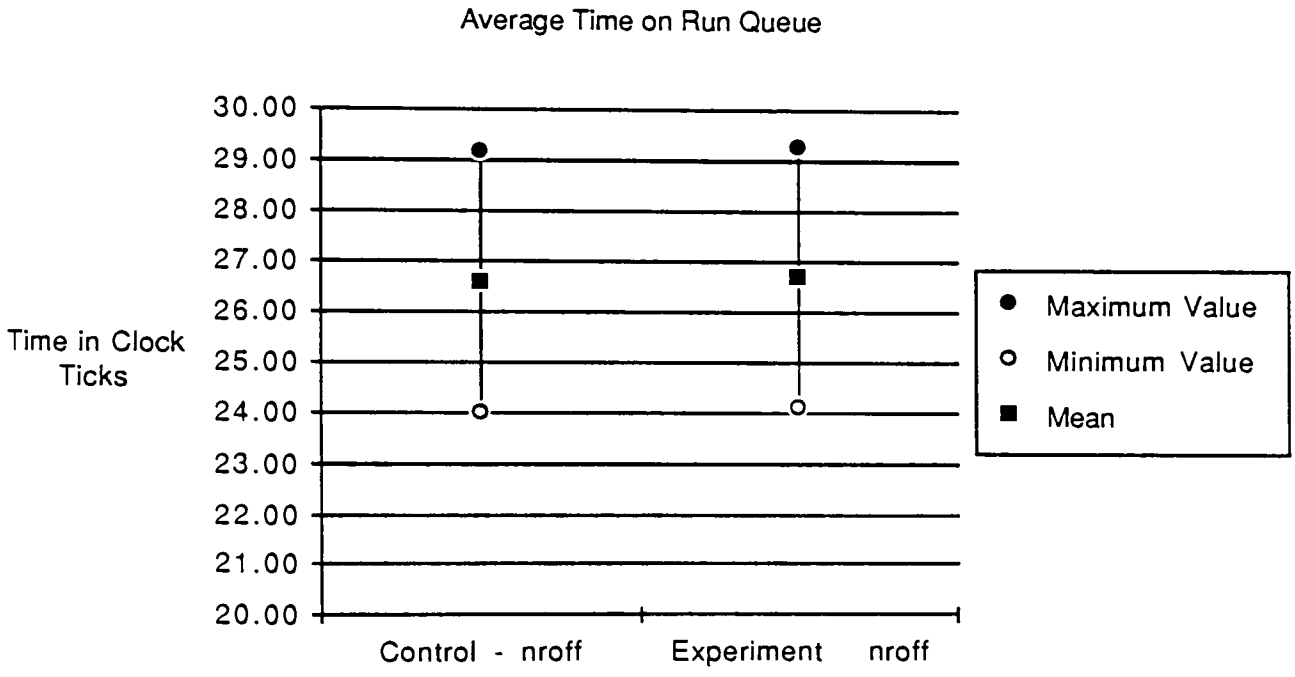

Average Time on Run Queue
Time in Clock Ticks
30.00
29.00
28.00
27.00
26.00
25.00
24.00
23.00
22.00
21.00
20.00
Control - nroff
Experiment nroff
Maximum Value
Minimum Value
Mean

Chart rq7

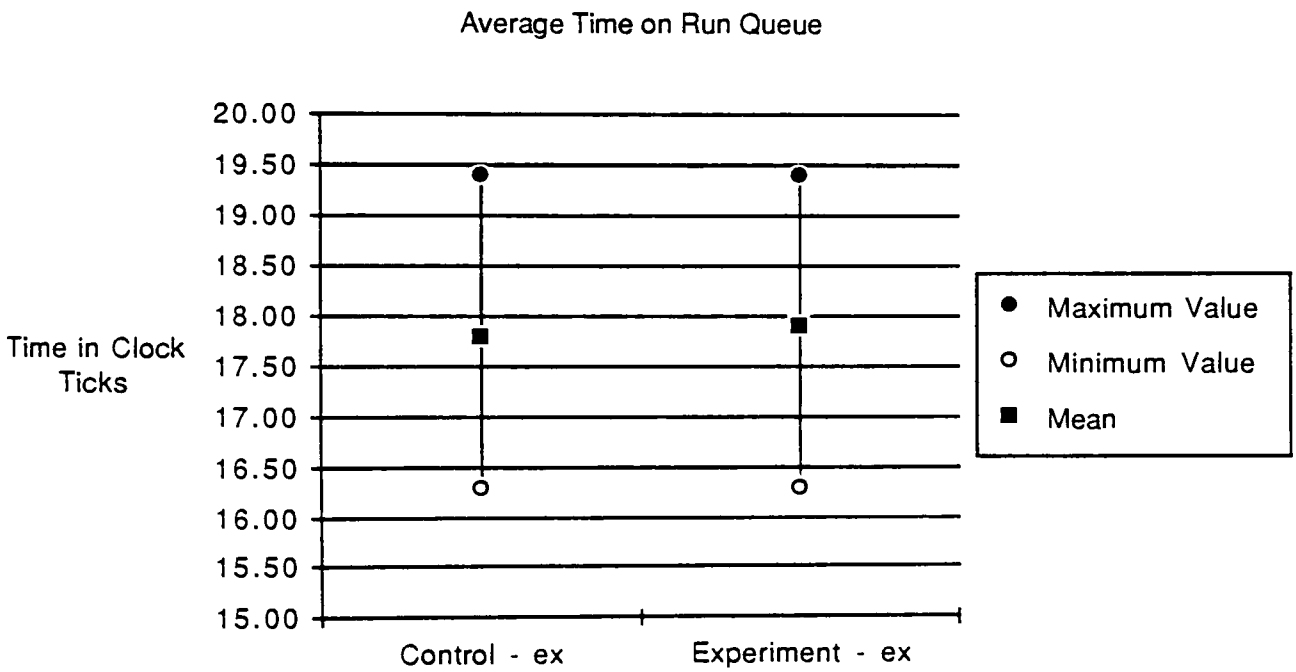

Average Time on Run Queue
Time in Clock Ticks
20.00
19.50
19.00
18.50
18.00
17.50
17.00
16.50
16.00
15.50
15.00
Control - ex
Experiment - ex
Maximum Value
Minimum Value
Mean

Chart rq8

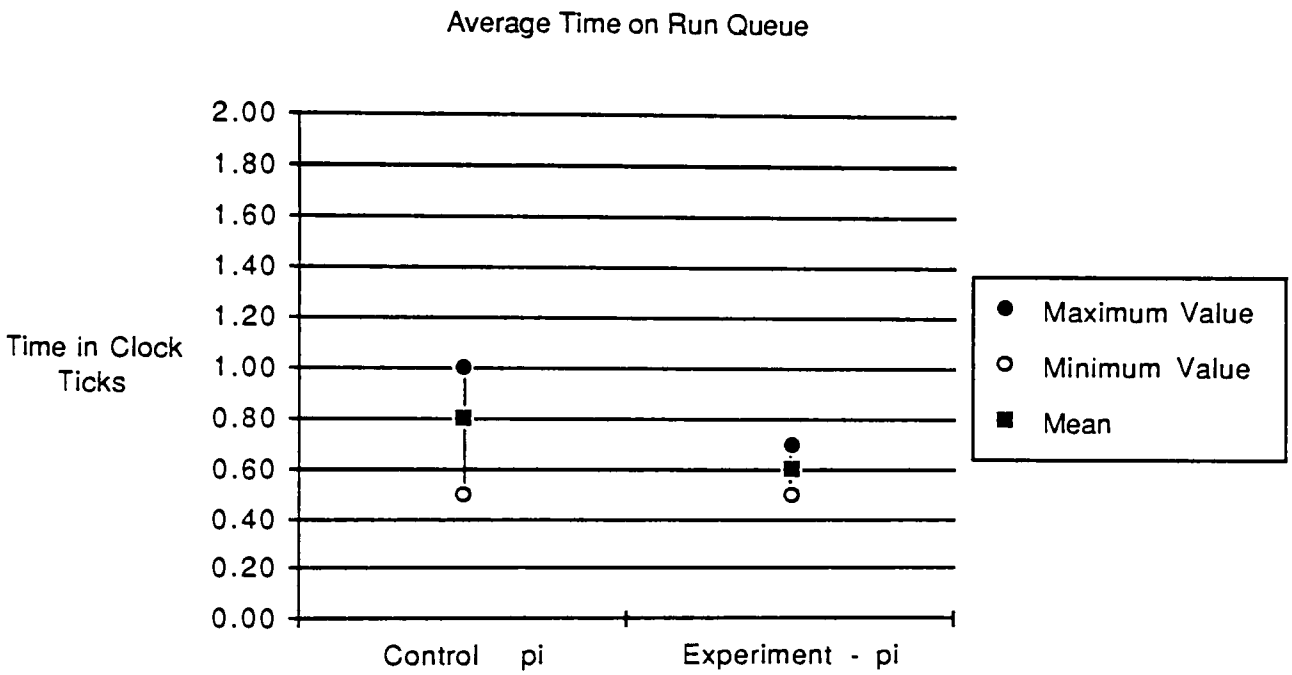

Average Time on Run Queue
Time in Clock Ticks
2.00
1.80
1.60
1.40
1.20
1.00
0.80
0.60
0.40
0.20
0.00
Control pi
Experiment - pi
Maximum Value
Minimum Value
Mean

Chart rq9

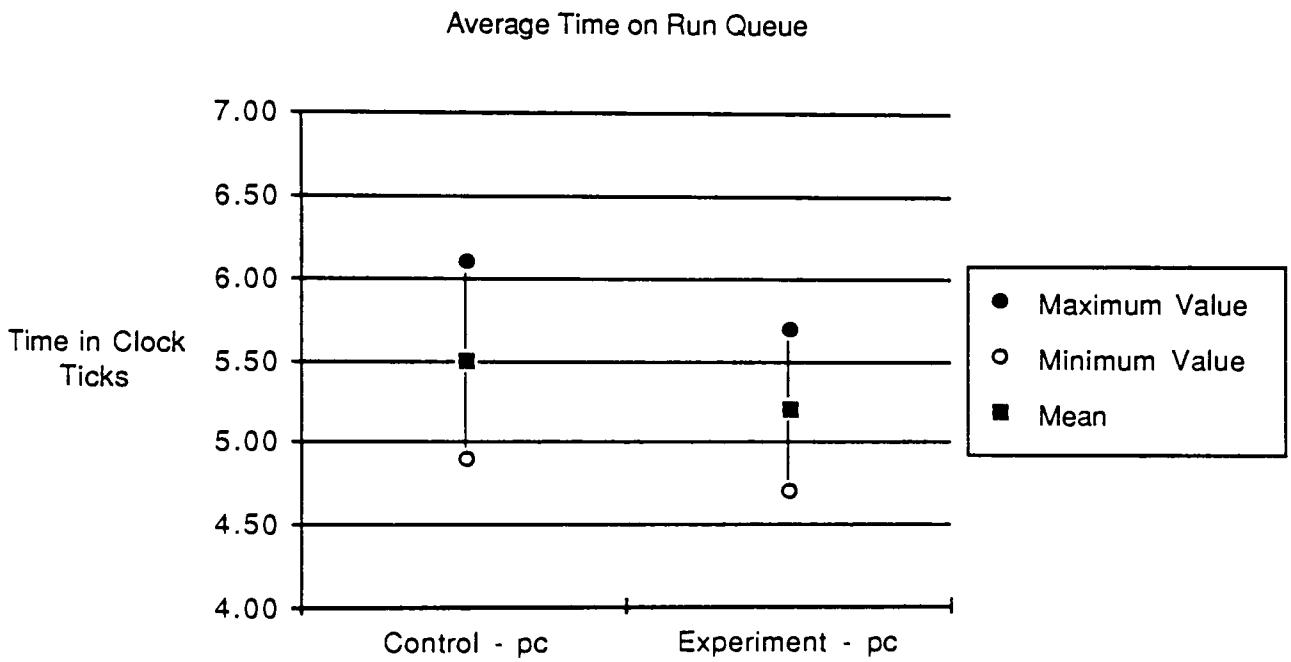
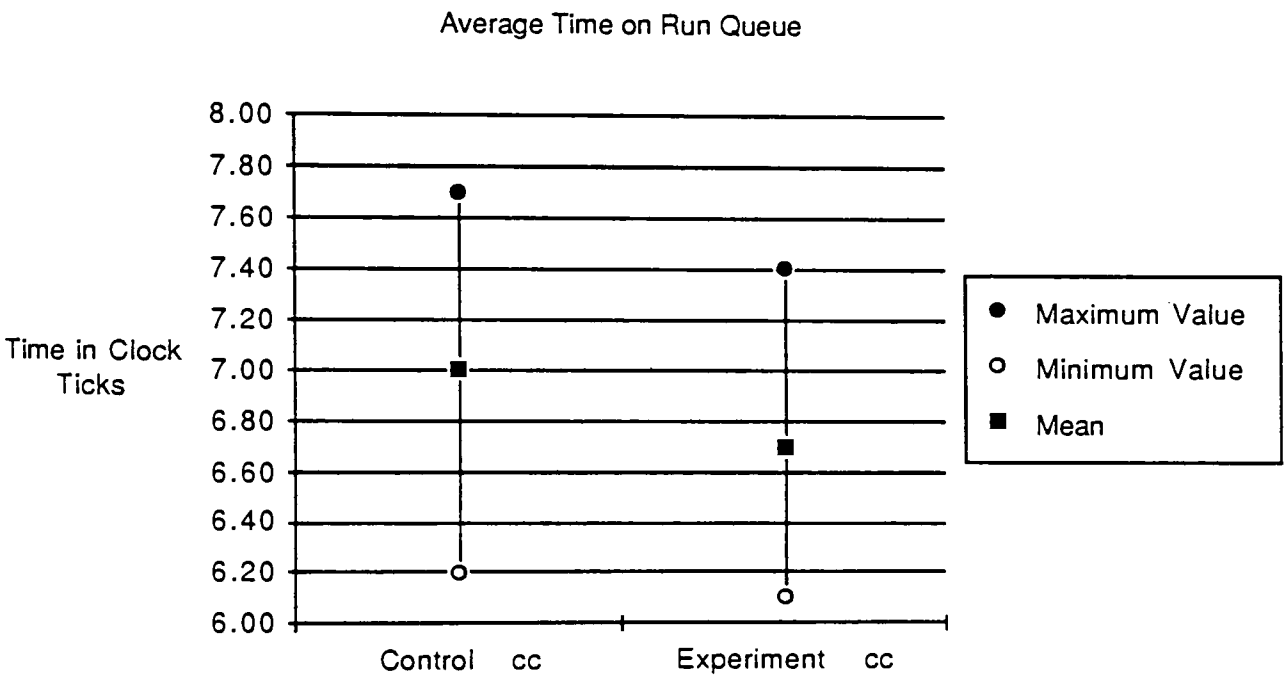

Average Time on Run Queue
Time in Clock Ticks
7.00
6.50
6.00
5.50
5.00
4.50
4.00
Control - pc
Experiment - pc
Maximum Value
Minimum Value
Mean

Chart rq10


Average Time on Run Queue
Time in Clock Ticks
8.00
7.80
7.60
7.40
7.20
7.00
6.80
6.60
6.40
6.20
6.00
Control cc
Experiment cc
Maximum Value
Minimum Value
Mean

Charts for the Changing Quantum Experiment

Introduction

The group of charts on the next several pages contains the results of the changing quantum experiment. This experiment examined the effect of assigning a process that has been chosen to run a definite quantum that is inversely proportional to its priority. Five variations of the same formula to calculate the quantum were tried. Each variation differed from the others only in the value of one variable which was assgined the values shown on the horizontal axis of each chart. Five sets of fifty runs each were performed. Charts seq1 to seq7 show the 90% confidence intervals of the mean elapsed time for each memeber of the family of processes. Charts sdq1 to sdq7 show the intervals one standard deviation on each side of the mean for each member of the family of processes.

Chart seq1

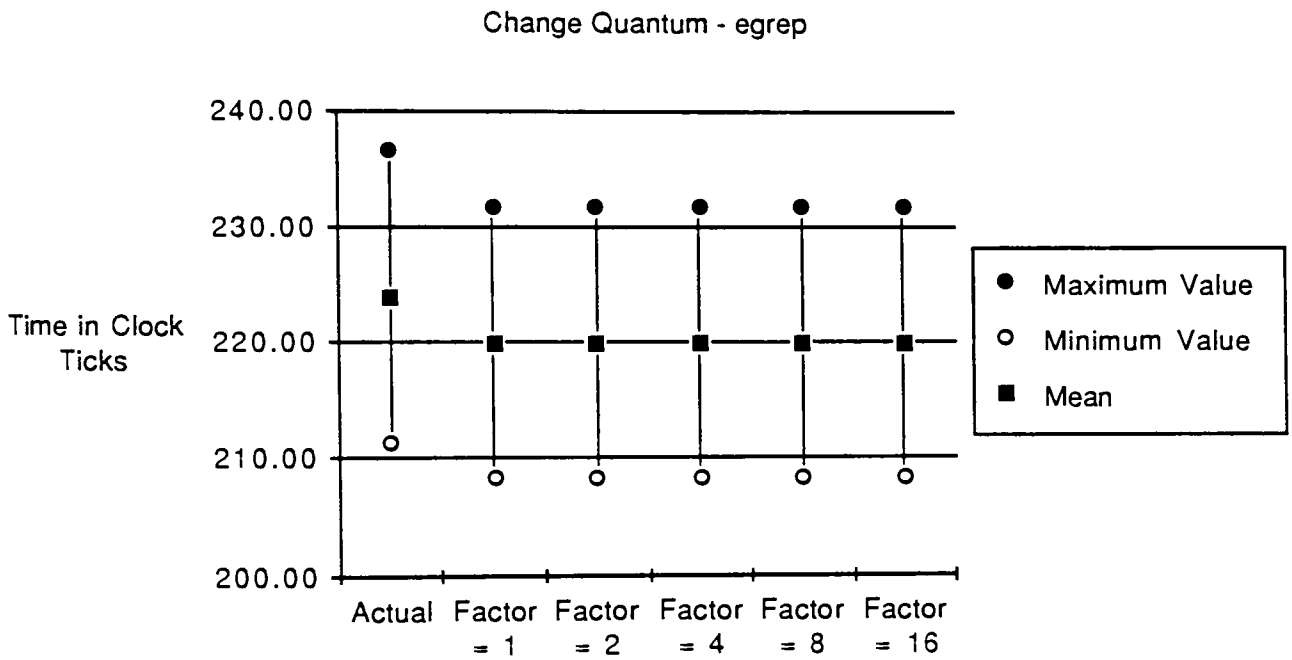

Change Quantum - egrep
Time in Clock Ticks
240.00
230.00
220.00
210.00
200.00
Actual
Factor = 1
Factor = 2
Factor = 4
Factor = 8
Factor = 16
Maximum Value
Minimum Value
Mean

Chart seq2

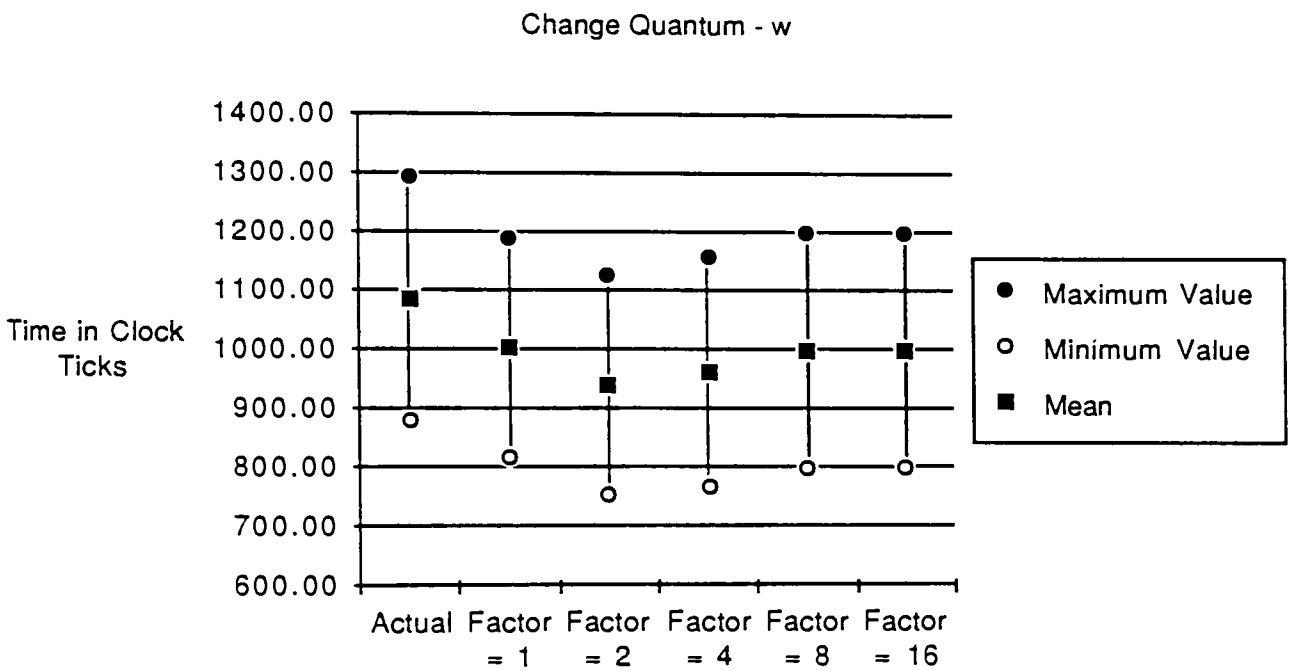

Change Quantum - w
Time in Clock Ticks
1400.00
1300.00
1200.00
1100.00
1000.00
900.00
800.00
700.00
600.00
Actual
Factor = 1
Factor = 2
Factor = 4
Factor = 8
Factor = 16
Maximum Value
Minimum Value
Mean

Chart seq3

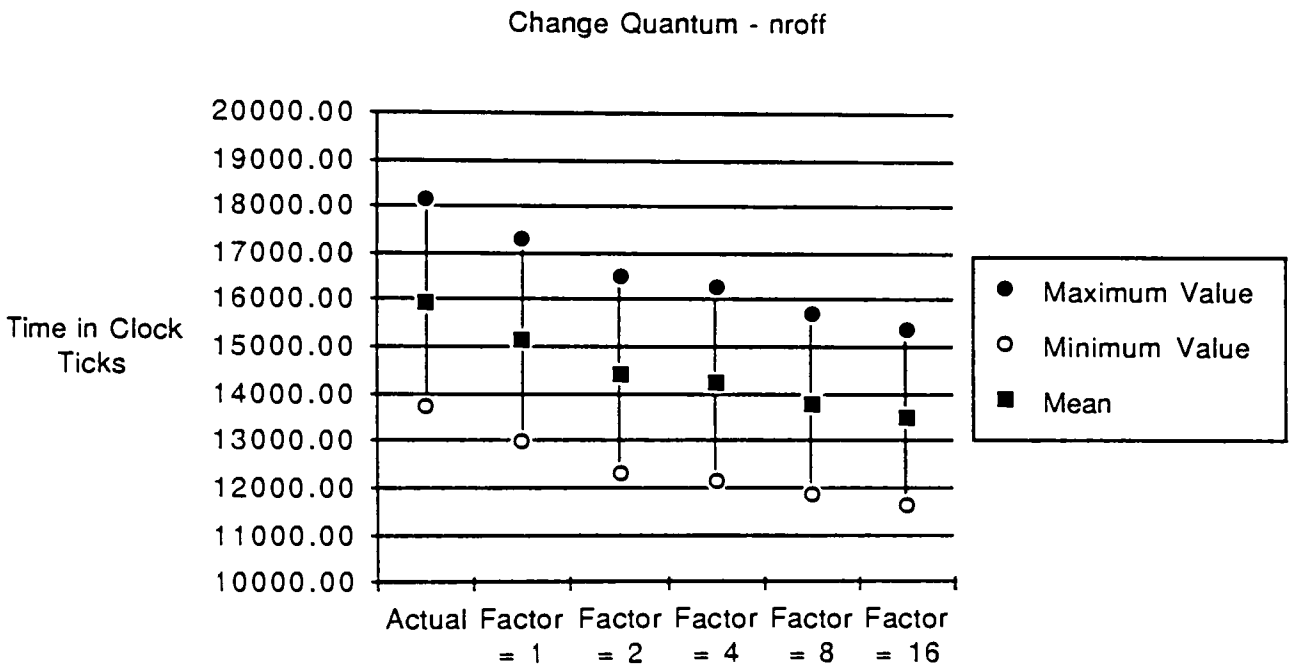

Change Quantum - nroff
Time in Clock Ticks
20000.00
19000.00
18000.00
17000.00
16000.00
15000.00
14000.00
13000.00
12000.00
11000.00
10000.00
Actual Factor = 1
Factor = 2
Factor = 4
Factor = 8
Factor = 16
Maximum Value
Minimum Value
Mean

Chart seq4

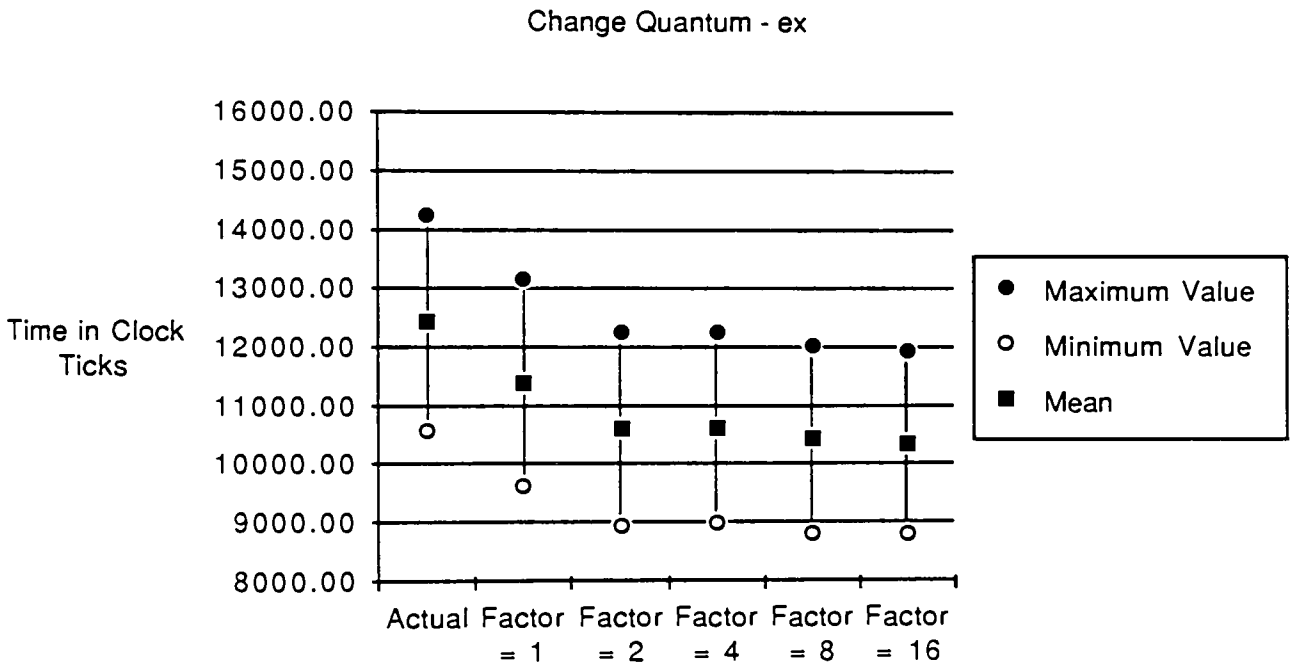

Change Quantum - ex
Time in Clock Ticks
16000.00
15000.00
14000.00
13000.00
12000.00
11000.00
10000.00
9000.00
8000.00
Actual Factor = 1
Factor = 2
Factor = 4
Factor = 8
Factor = 16
Maximum Value
Minimum Value
Mean

Chart seq5

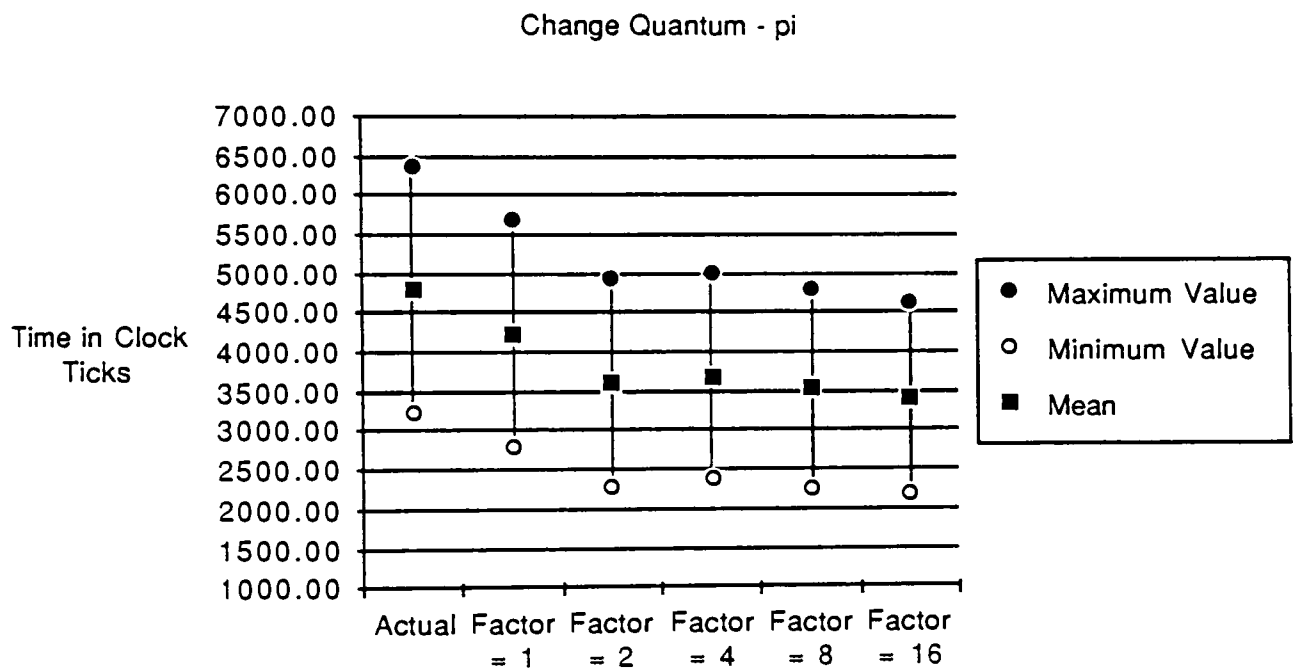

Change Quantum - pi
Time in Clock Ticks
7000.00
6500.00
6000.00
5500.00
5000.00
4500.00
4000.00
3500.00
3000.00
2500.00
2000.00
1500.00
1000.00
Actual Factor = 1
Factor = 2
Factor = 4
Factor = 8
Factor = 16
Maximum Value
Minimum Value
Mean

Chart seq6

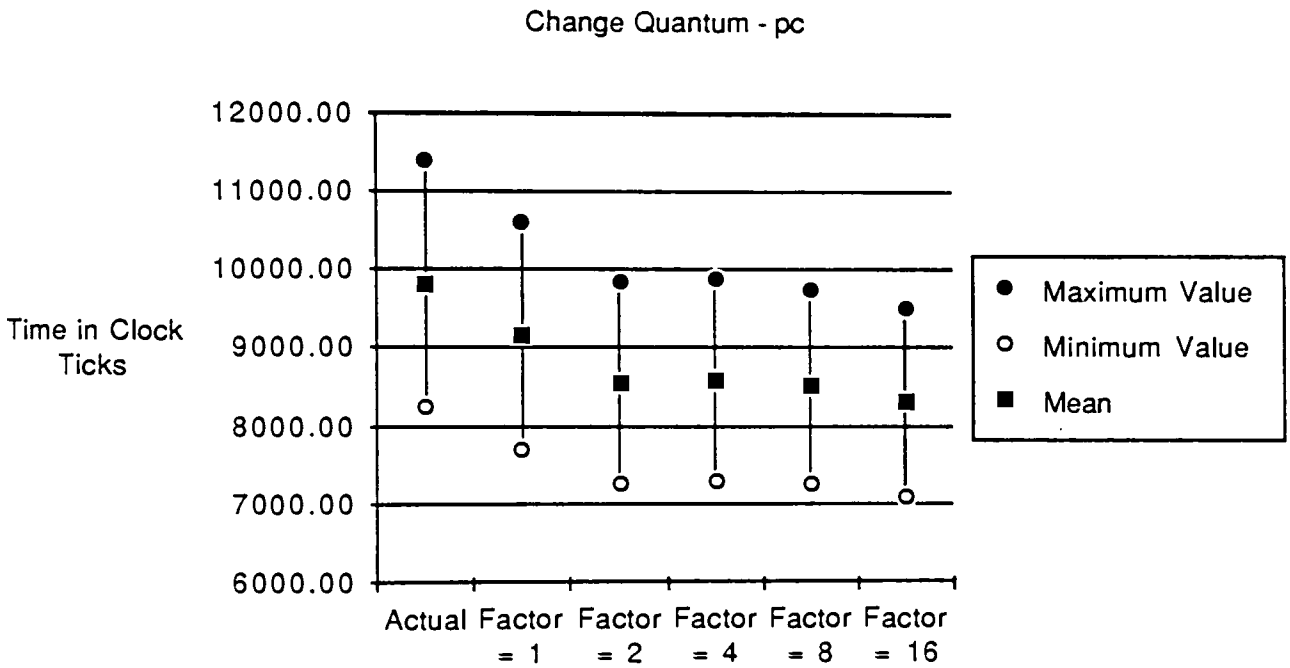

Change Quantum - pc
Time in Clock Ticks
12000.00
11000.00
10000.00
9000.00
8000.00
7000.00
6000.00
Actual Factor = 1
Factor = 2
Factor = 4
Factor = 8
Factor = 16
Maximum Value
Minimum Value
Mean

Chart seq7

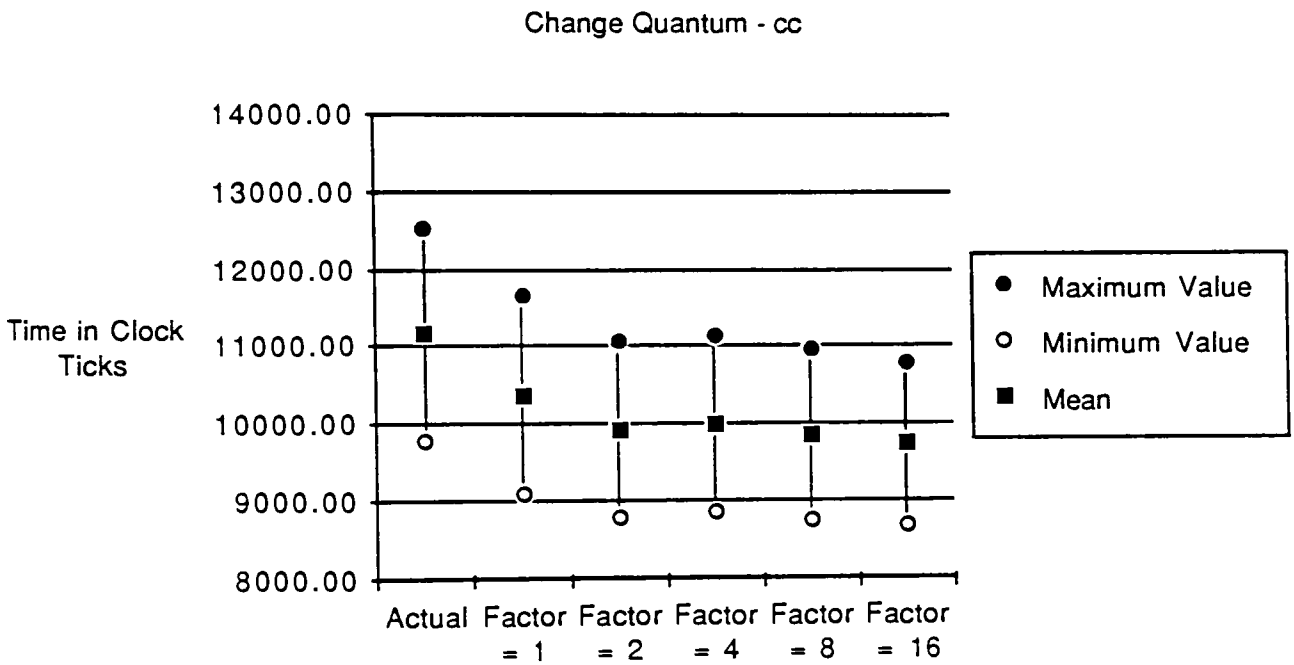

Change Quantum - cc
Time in Clock Ticks
14000.00
13000.00
12000.00
11000.00
10000.00
9000.00
8000.00
Actual Factor = 1
Factor = 2
Factor = 4
Factor = 8
Factor = 16
Maximum Value
Minimum Value
Mean

Chart sdq1

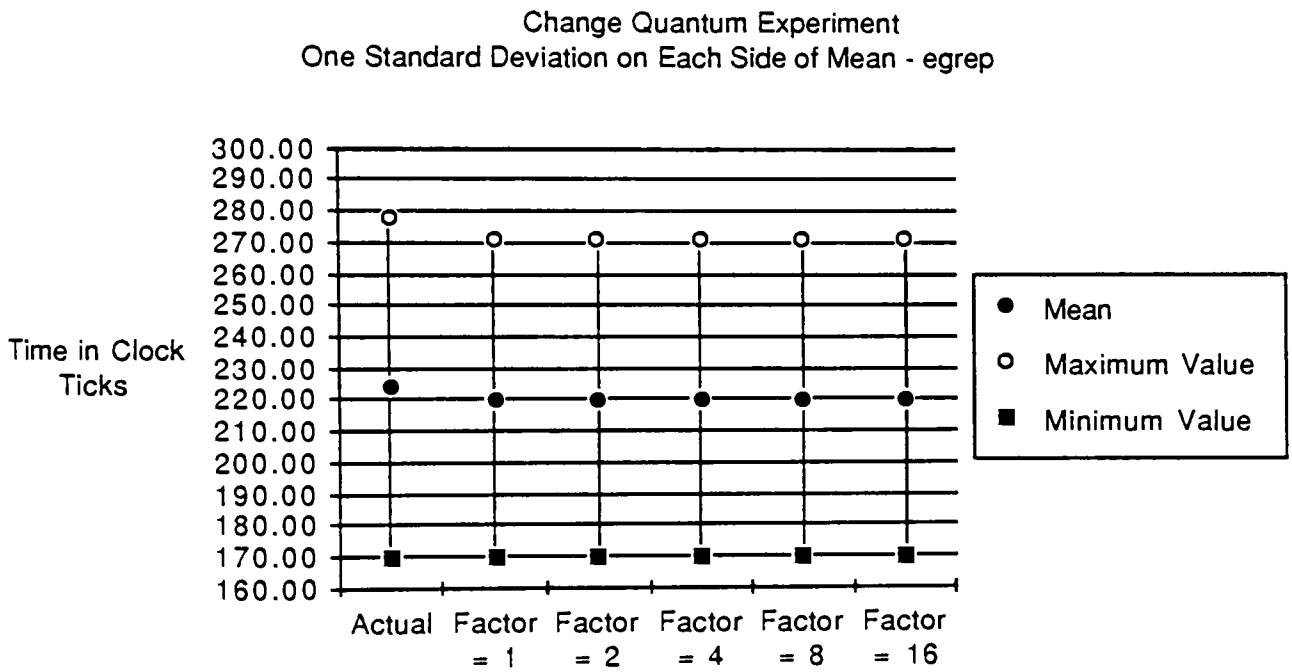

Change Quantum Experiment
One Standard Deviation on Each Side of Mean - egrep
Time in Clock Ticks
300.00
290.00
280.00
270.00
260.00
250.00
240.00
230.00
220.00
210.00
200.00
190.00
180.00
170.00
160.00
Actual
Factor = 1
Factor = 2
Factor = 4
Factor = 8
Factor = 16
Mean
Maximum Value
Minimum Value

Chart sdq2

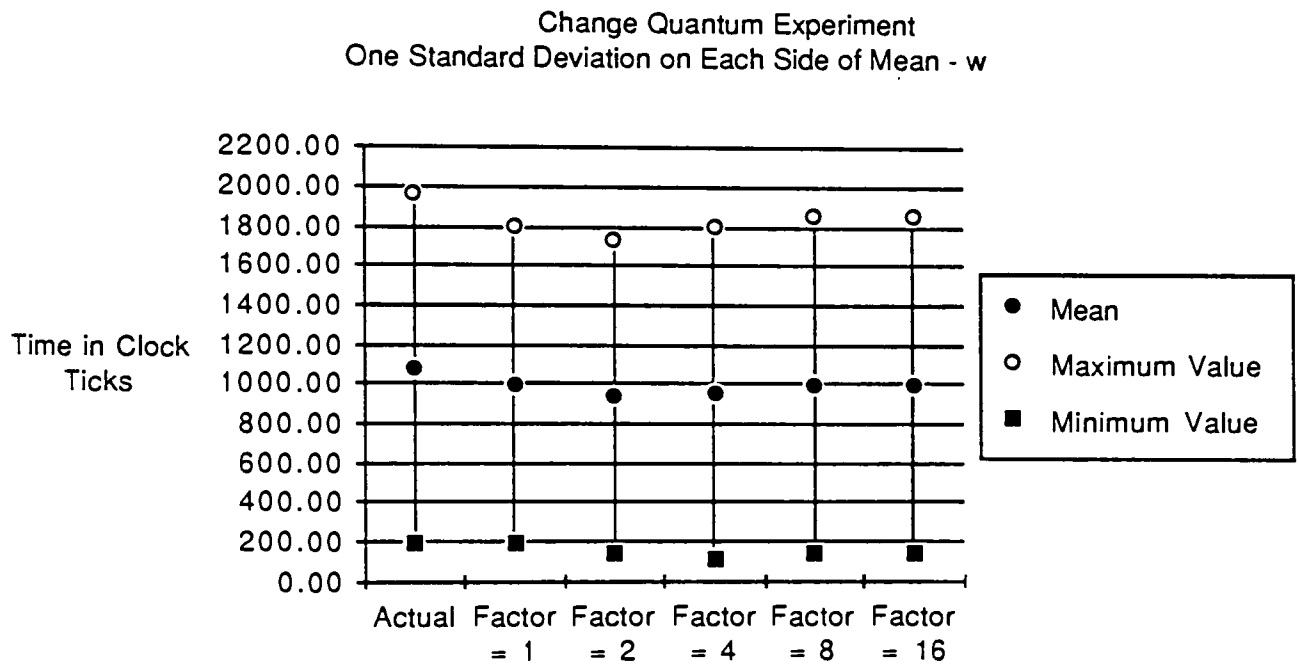

Change Quantum Experiment
One Standard Deviation on Each Side of Mean - w
Time in Clock Ticks
2200.00
2000.00
1800.00
1600.00
1400.00
1200.00
1000.00
800.00
600.00
400.00
200.00
0.00
Actual
Factor = 1
Factor = 2
Factor = 4
Factor = 8
Factor = 16
Mean
Maximum Value
Minimum Value

Chart sdq3

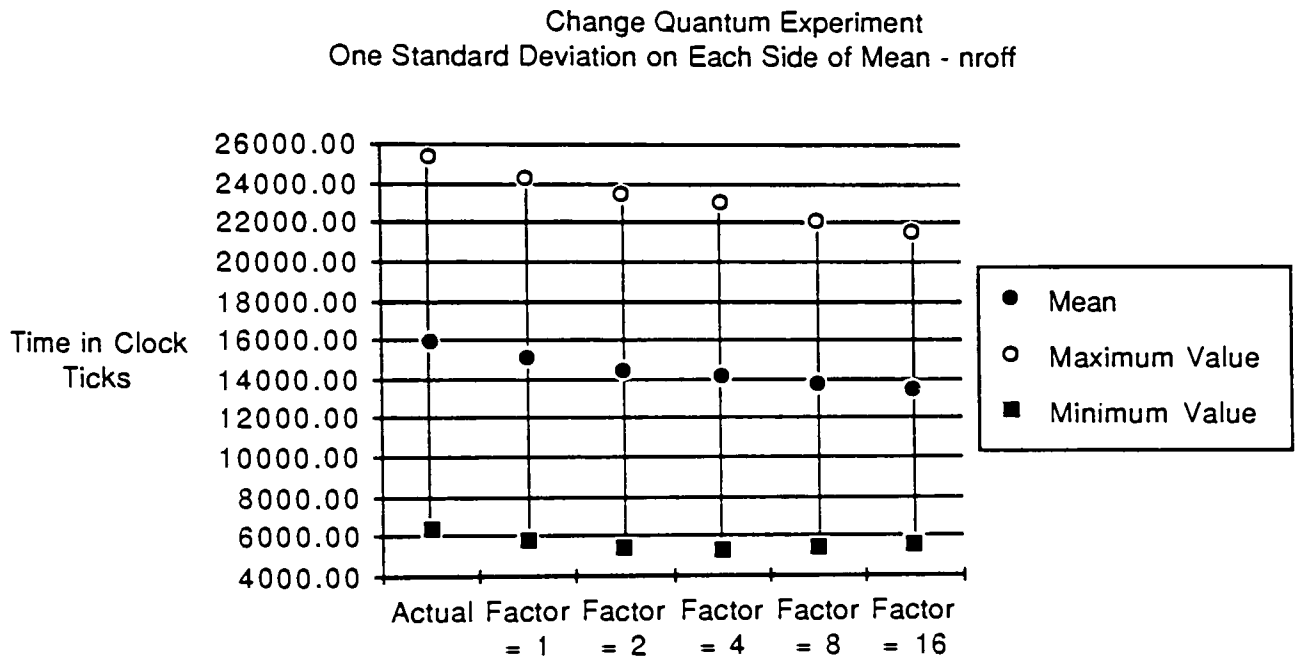

Change Quantum Experiment
One Standard Deviation on Each Side of Mean - nroff
Time in Clock Ticks
26000.00
24000.00
22000.00
20000.00
18000.00
16000.00
14000.00
12000.00
10000.00
8000.00
6000.00
4000.00
Actual Factor = 1
Factor = 2
Factor = 4
Factor = 8
Factor = 16
Mean
Maximum Value
Minimum Value

Chart sdq4

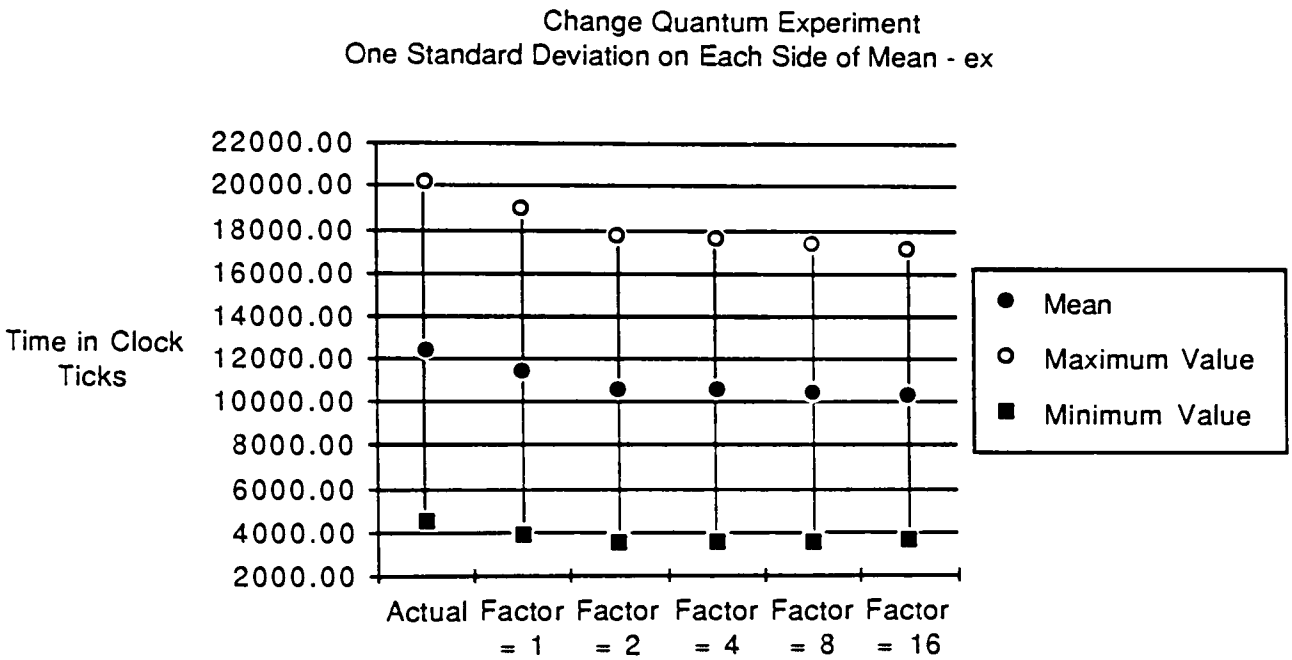

Change Quantum Experiment
One Standard Deviation on Each Side of Mean - ex
Time in Clock Ticks
22000.00
20000.00
18000.00
16000.00
14000.00
12000.00
10000.00
8000.00
6000.00
4000.00
2000.00
Actual Factor = 1
Factor = 2
Factor = 4
Factor = 8
Factor = 16
Mean
Maximum Value
Minimum Value

Chart sdq5

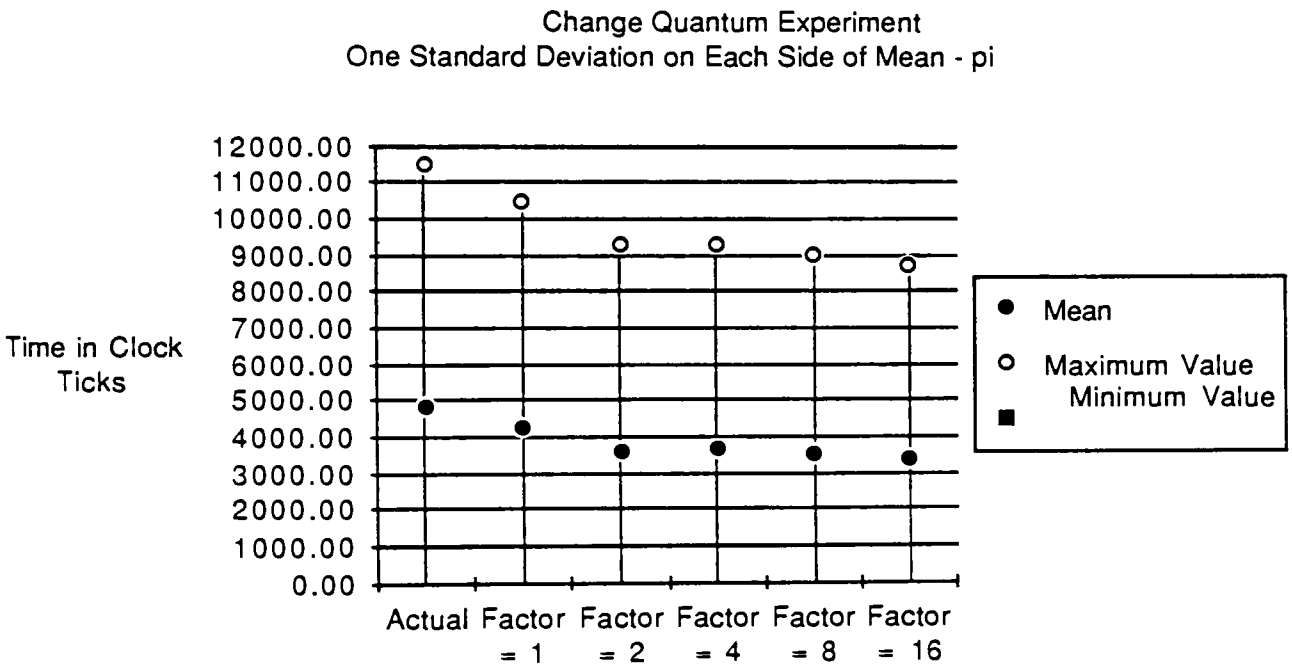

Change Quantum Experiment
One Standard Deviation on Each Side of Mean - pi
Time in Clock Ticks
12000.00
11000.00
10000.00
9000.00
8000.00
7000.00
6000.00
5000.00
4000.00
3000.00
2000.00
1000.00
0.00
Actual Factor = 1
Factor = 2
Factor = 4
Factor = 8
Factor = 16
Mean
Maximum Value
Minimum Value

Chart sdq6

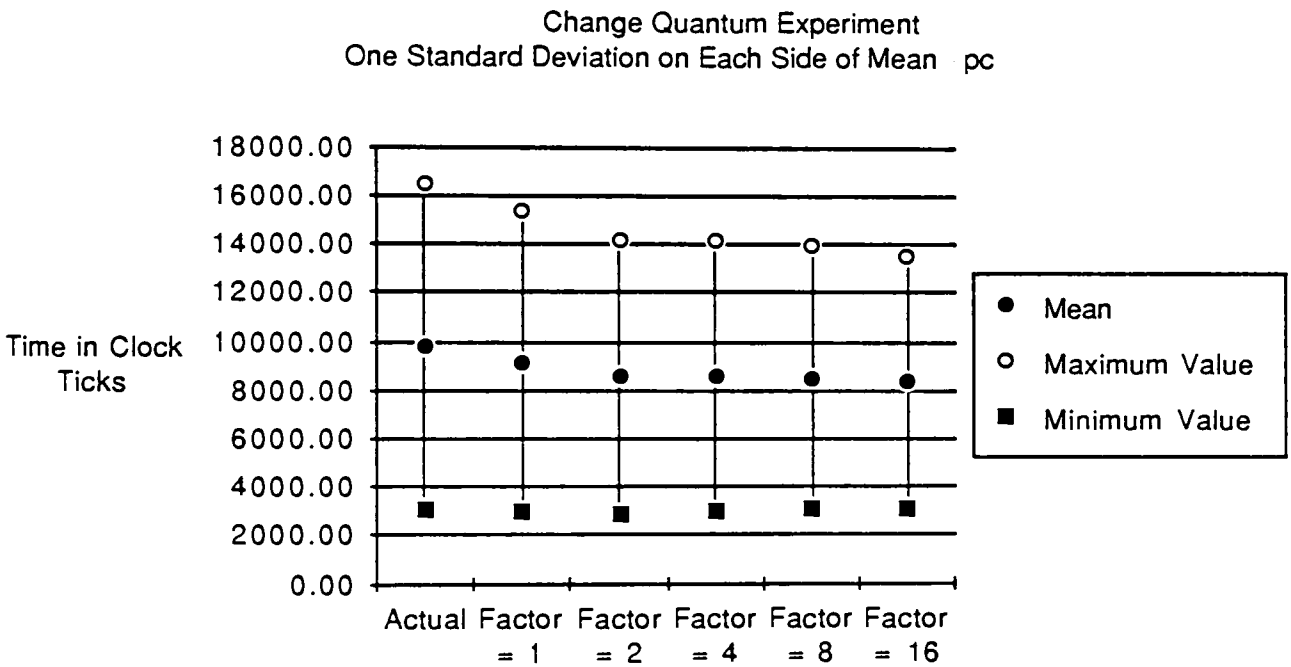
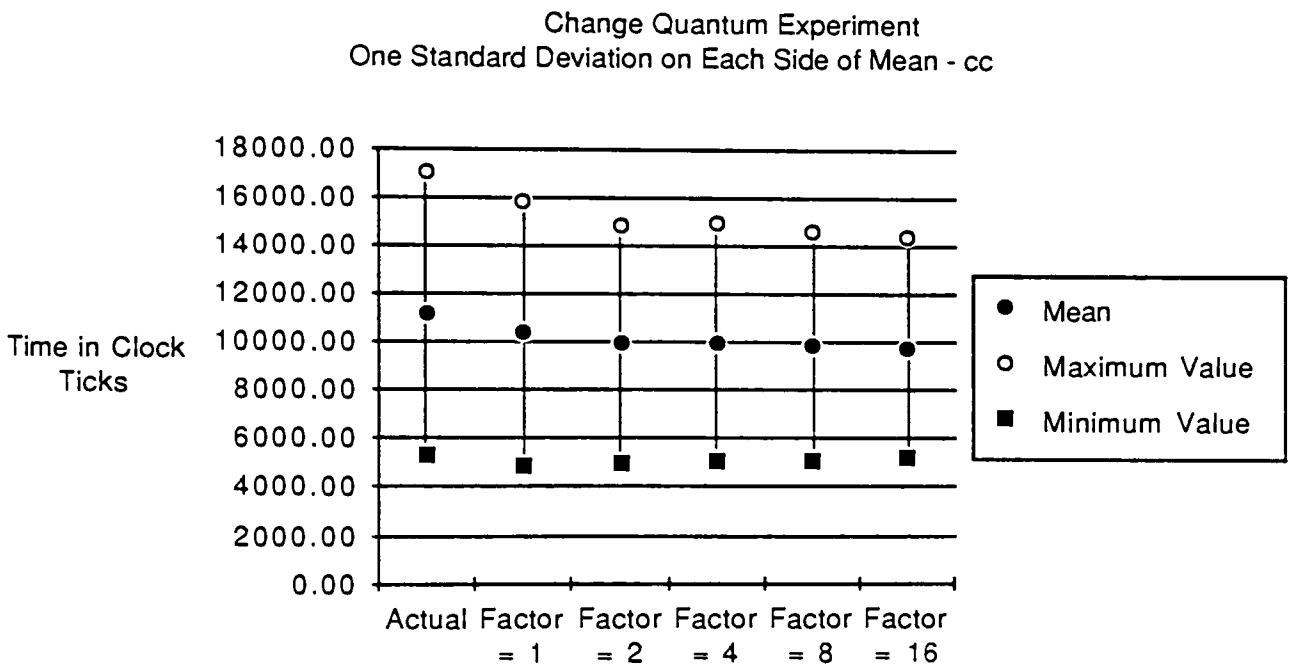

Change Quantum Experiment
One Standard Deviation on Each Side of Mean pc
Time in Clock Ticks
18000.00
16000.00
14000.00
12000.00
10000.00
8000.00
6000.00
4000.00
2000.00
0.00
Actual Factor = 1
Factor = 2
Factor = 4
Factor = 8
Factor = 16
Mean
Maximum Value
Minimum Value

Chart sdq7


Change Quantum Experiment
One Standard Deviation on Each Side of Mean - cc
Time in Clock Ticks
18000.00
16000.00
14000.00
12000.00
10000.00
8000.00
6000.00
4000.00
2000.00
0.00
Actual Factor = 1
Factor = 2
Factor = 4
Factor = 8
Factor = 16
Mean
Maximum Value
Minimum Value

Charts for the Changing the Number of Queues Experiment

Introduction

The following set of charts illustrates the results of the experiment in which the number of queues in the scheduler was changed. This experiment involved runs of the model with one, eight, sixteen, sixty-four, and one hundred twenty-eight queues. The charts labelled secq1 to secq7 contain the 90% confidence intervals of the mean elapsed time for each member of the family of processes in each set of runs. The number of queues for each confidence interval is noted on the horizontal axis directly below that confidence interval. Charts sdcq1 tio sdcq7 indicate the interval one standard deviation on each side of the mean for each member of the family of processes. Here too the number of queues corresponding to a given interval on the chart is noted just below that interval on the horizontal axis.

Chart secq1

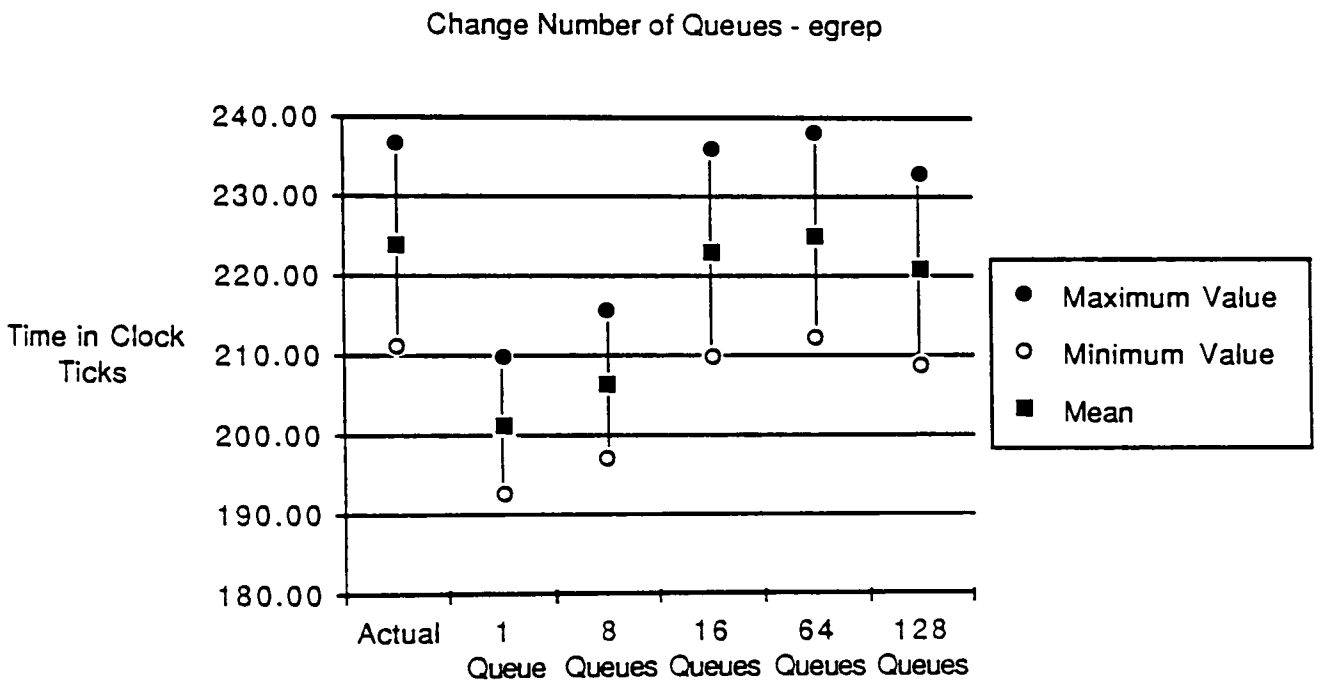

Change Number of Queues - egrep
Time in Clock Ticks
240.00
230.00
220.00
210.00
200.00
190.00
180.00
Actual
1 Queue
8 Queues
16 Queues
64 Queues
128 Queues
Maximum Value
Minimum Value
Mean

Chart secq2

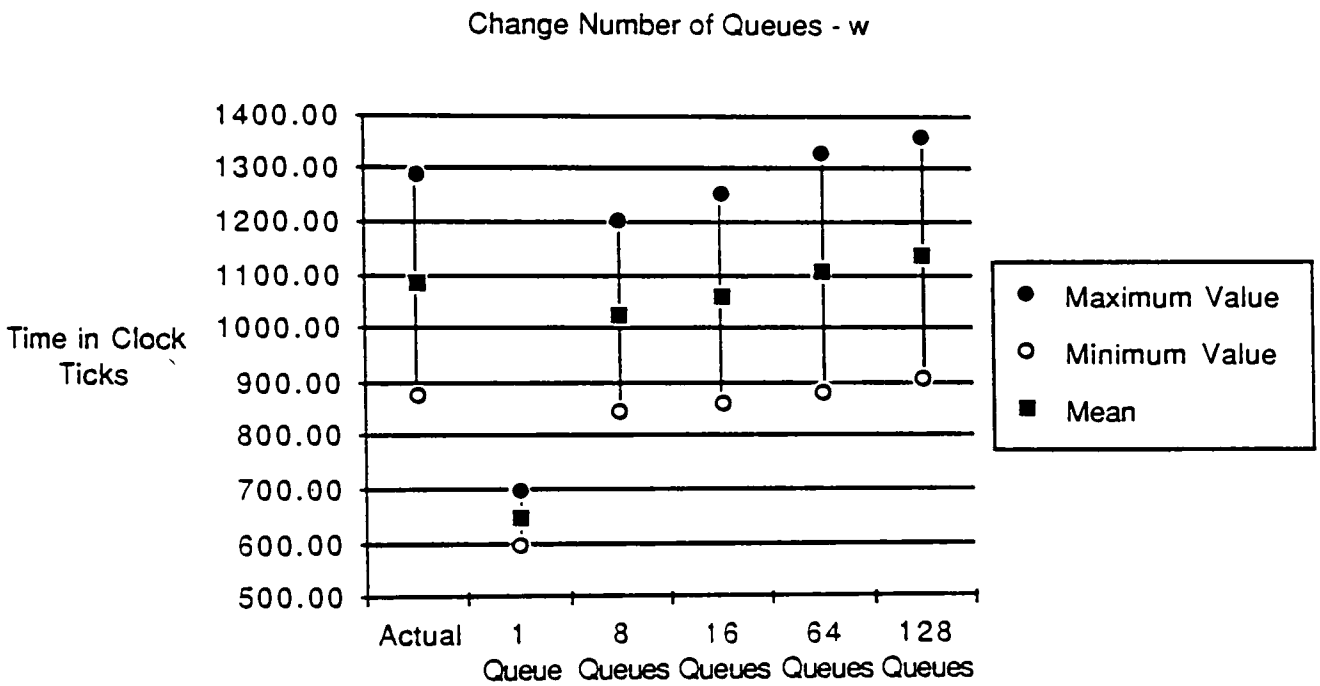

Change Number of Queues - w
Time in Clock Ticks
1400.00
1300.00
1200.00
1100.00
1000.00
900.00
800.00
700.00
600.00
500.00
Actual
1 Queue
8 Queues
16 Queues
64 Queues
128 Queues
Maximum Value
Minimum Value
Mean

Chart secq3

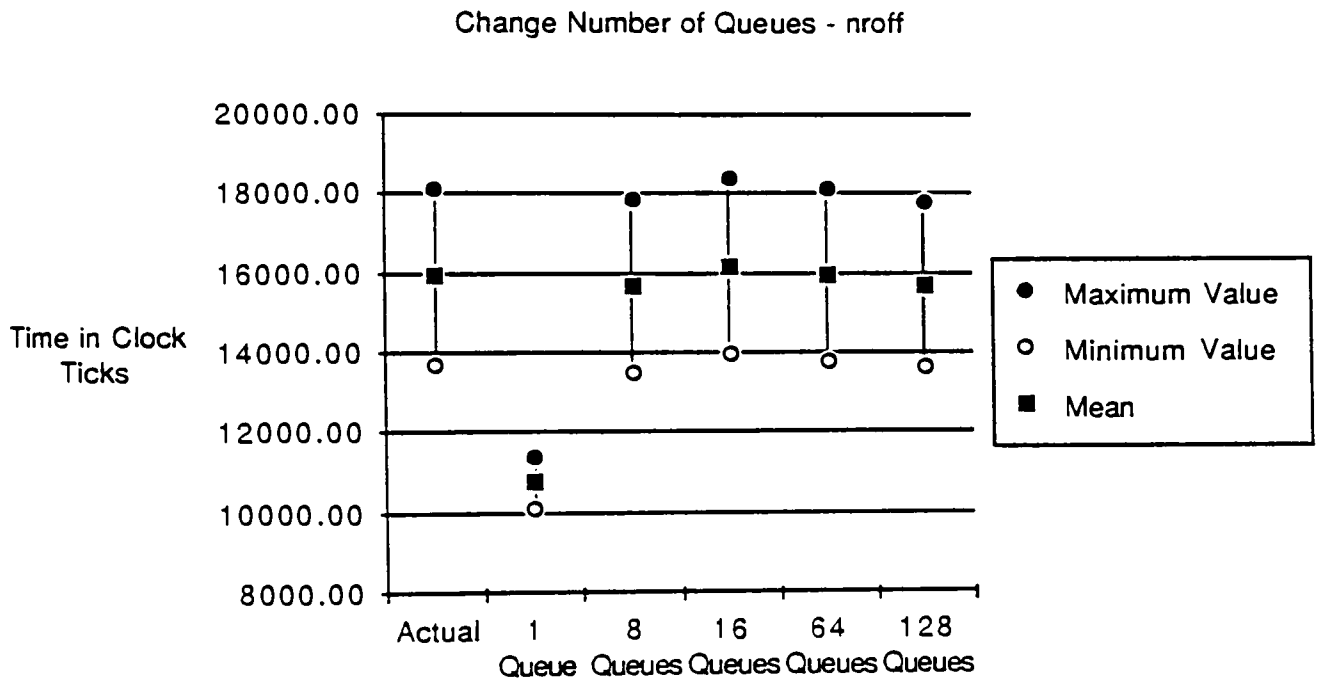

Change Number of Queues - nroff
Time in Clock Ticks
20000.00
18000.00
16000.00
14000.00
12000.00
10000.00
8000.00
Actual
1 Queue
8 Queues
16 Queues
64 Queues
128 Queues
Maximum Value
Minimum Value
Mean

Chart secq4

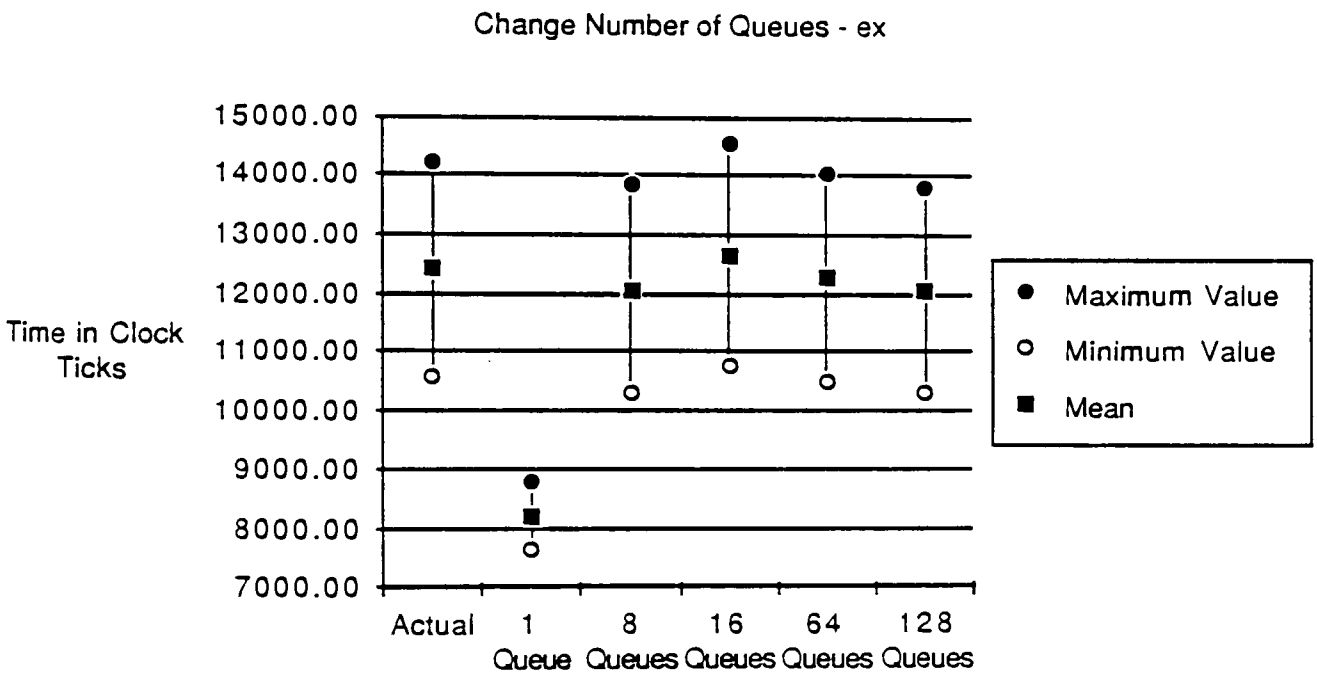

Change Number of Queues - ex
Time in Clock Ticks
15000.00
14000.00
13000.00
12000.00
11000.00
10000.00
9000.00
8000.00
7000.00
Actual
1 Queue
8 Queues
16 Queues
64 Queues
128 Queues
Maximum Value
Minimum Value
Mean

Chart secq5

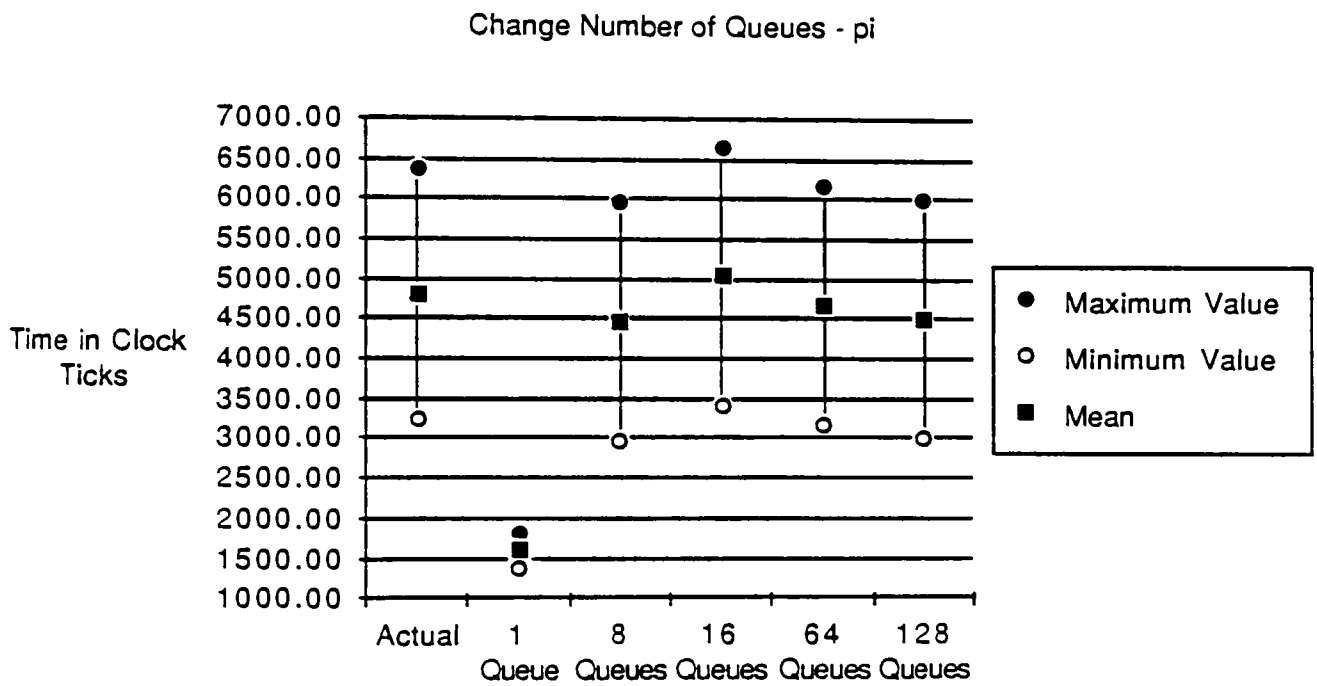

Change Number of Queues - pi
Time in Clock Ticks
7000.00
6500.00
6000.00
5500.00
5000.00
4500.00
4000.00
3500.00
3000.00
2500.00
2000.00
1500.00
1000.00
Actual
1 Queue
8 Queues
16 Queues
64 Queues
128 Queues
Maximum Value
Minimum Value
Mean

Chart secq6

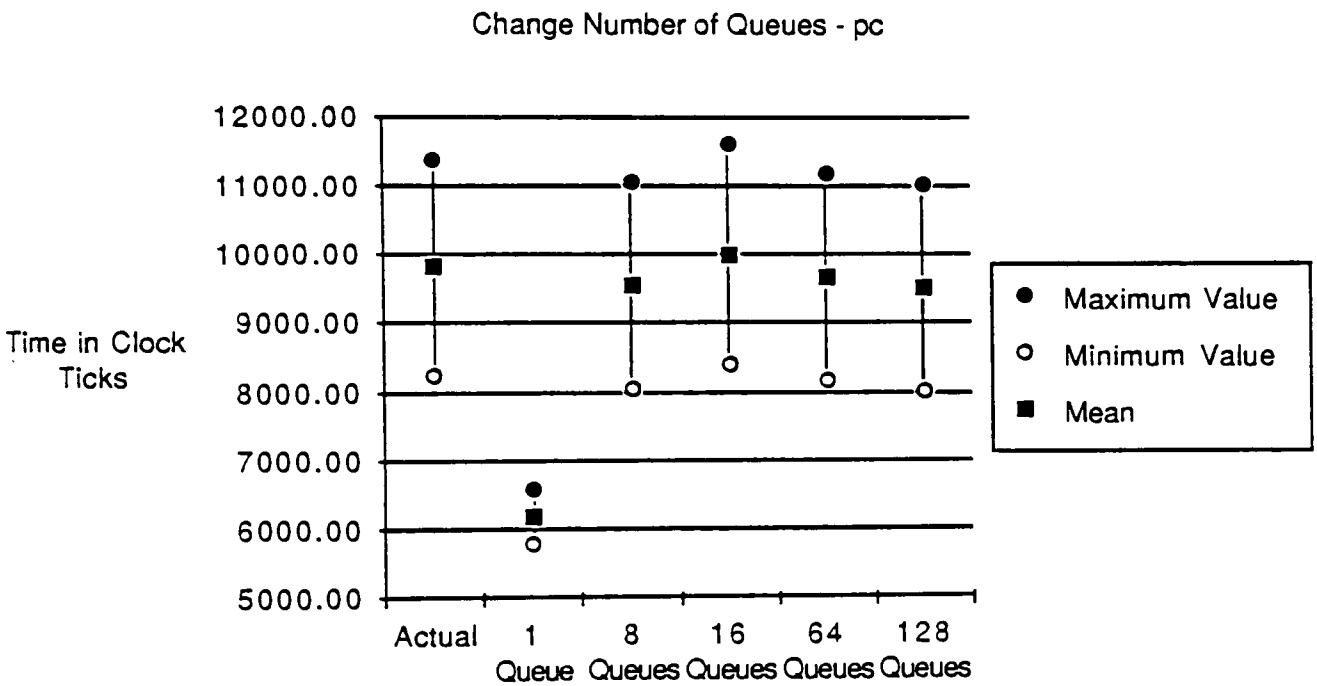

Change Number of Queues - pc
Time in Clock Ticks
12000.00
11000.00
10000.00
9000.00
8000.00
7000.00
6000.00
5000.00
Actual
1 Queue
8 Queues
16 Queues
64 Queues
128 Queues
Maximum Value
Minimum Value
Mean

Chart secq7

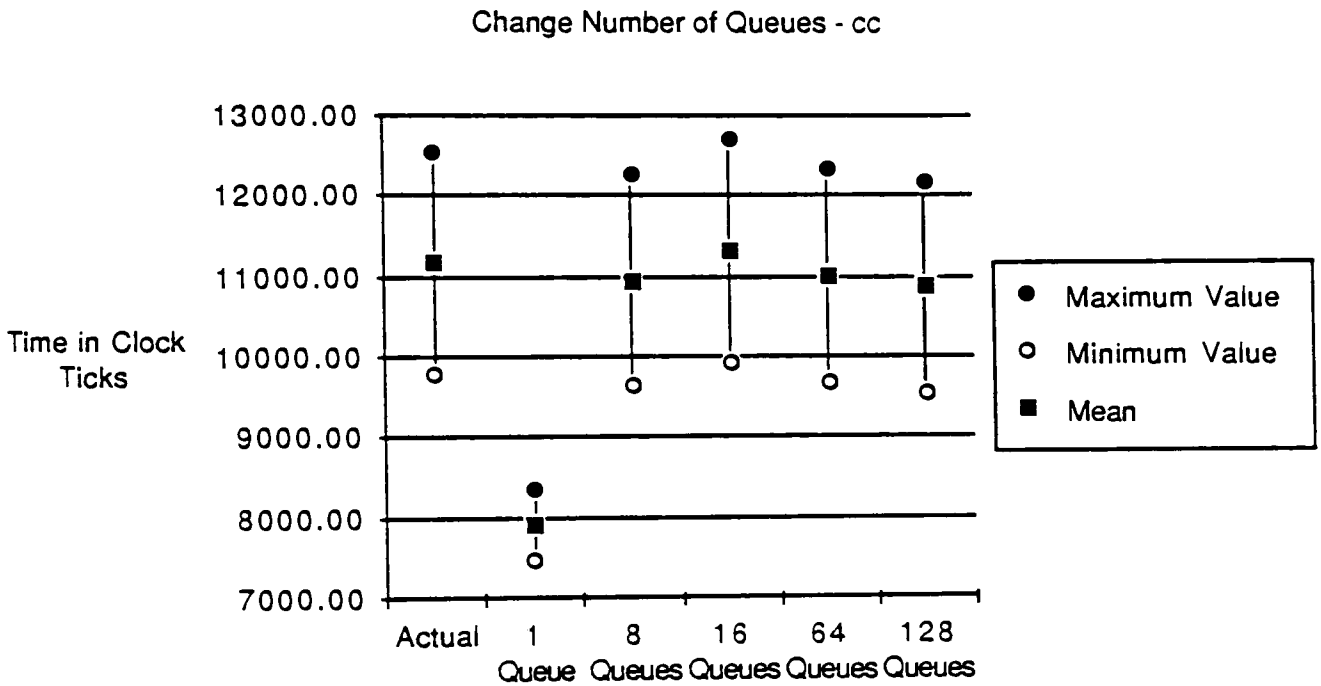

Change Number of Queues - cc
Time in Clock Ticks
13000.00
12000.00
11000.00
10000.00
9000.00
8000.00
7000.00
Actual
1 Queue
8 Queues
16 Queues
64 Queues
128 Queues
Maximum Value
Minimum Value
Mean

Chart sdcq1

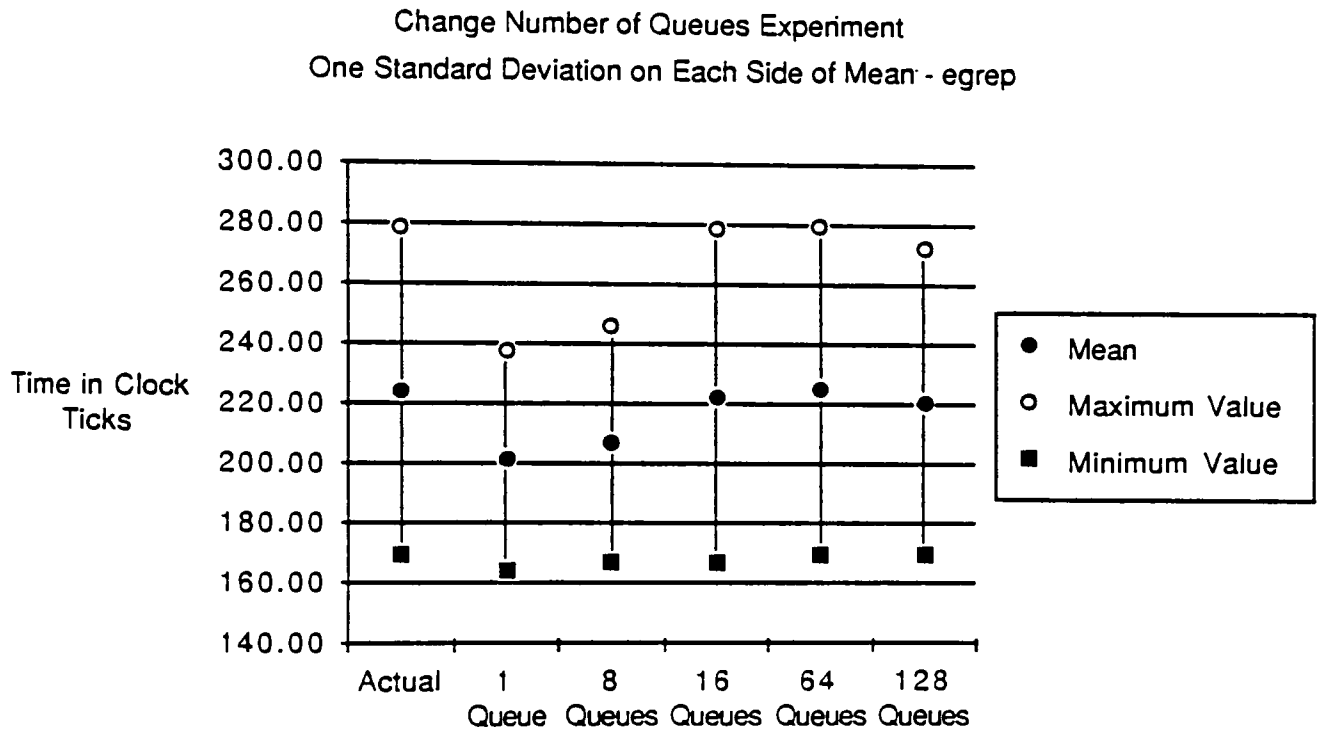

Change Number of Queues Experiment
One Standard Deviation on Each Side of Mean - egrep
Time in Clock Ticks
300.00
280.00
260.00
240.00
220.00
200.00
180.00
160.00
140.00
Actual
1 Queue
8 Queues
16 Queues
64 Queues
128 Queues
Mean
Maximum Value
Minimum Value

Chart sdcq2

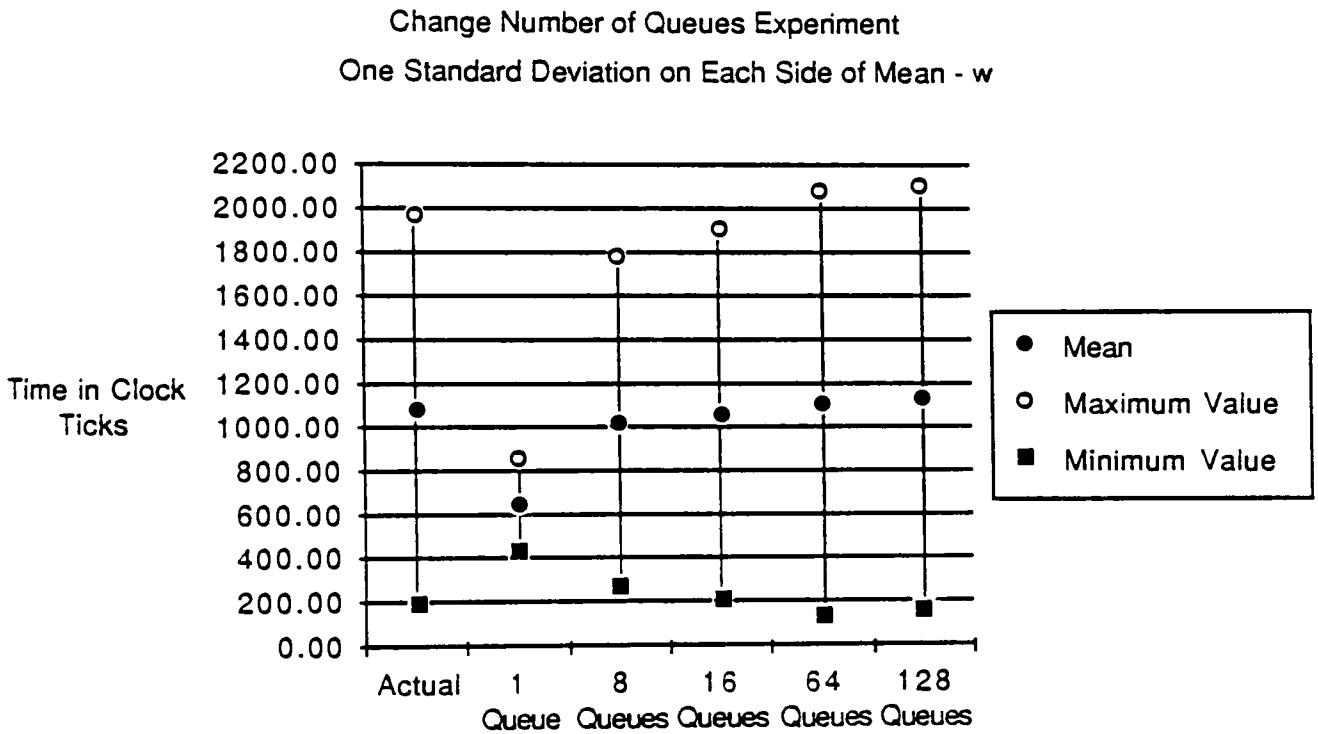

Change Number of Queues Experiment
One Standard Deviation on Each Side of Mean - w
Time in Clock Ticks
2200.00
2000.00
1800.00
1600.00
1400.00
1200.00
1000.00
800.00
600.00
400.00
200.00
0.00
Actual
1 Queue
8 Queues
16 Queues
64 Queues
128 Queues
Mean
Maximum Value
Minimum Value

Chart sdcq3

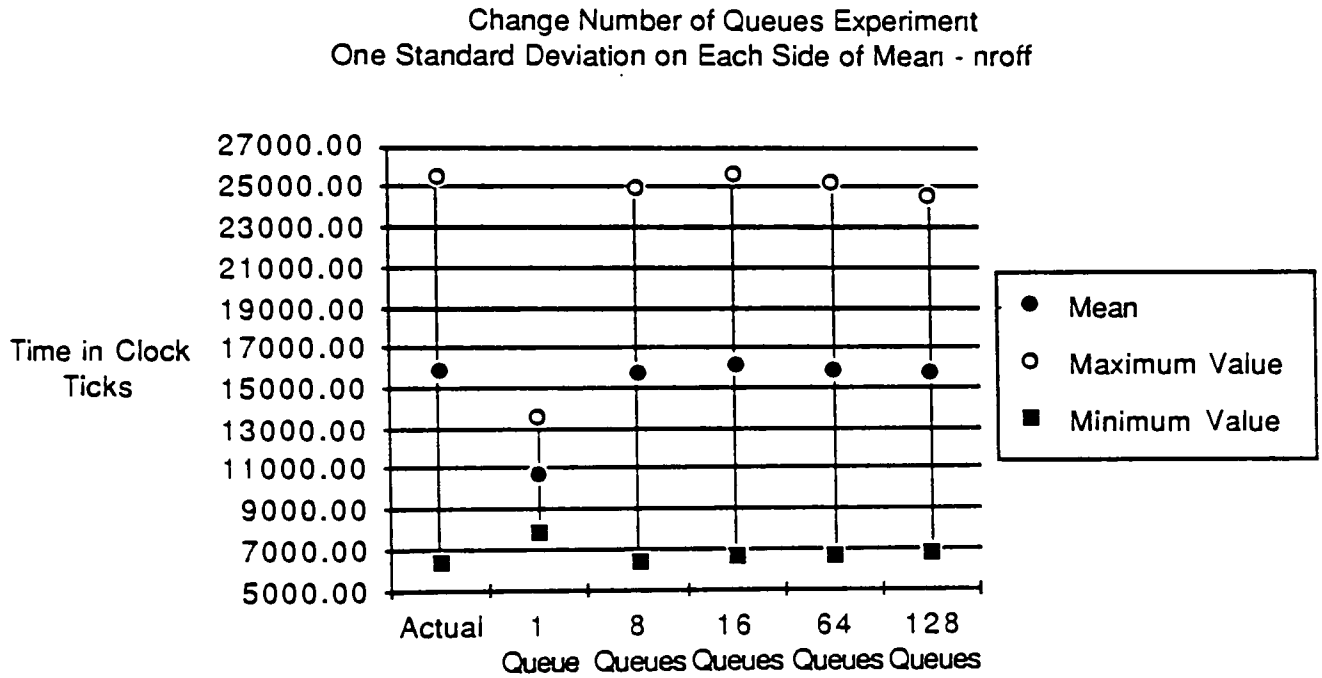

Change Number of Queues Experiment
One Standard Deviation on Each Side of Mean - nroff
Time in Clock Ticks
27000.00
25000.00
23000.00
21000.00
19000.00
17000.00
15000.00
13000.00
11000.00
9000.00
7000.00
5000.00
Actual
1 Queue
8 Queues
16 Queues
64 Queues
128 Queues
Mean
Maximum Value
Minimum Value

Chart sdcq4

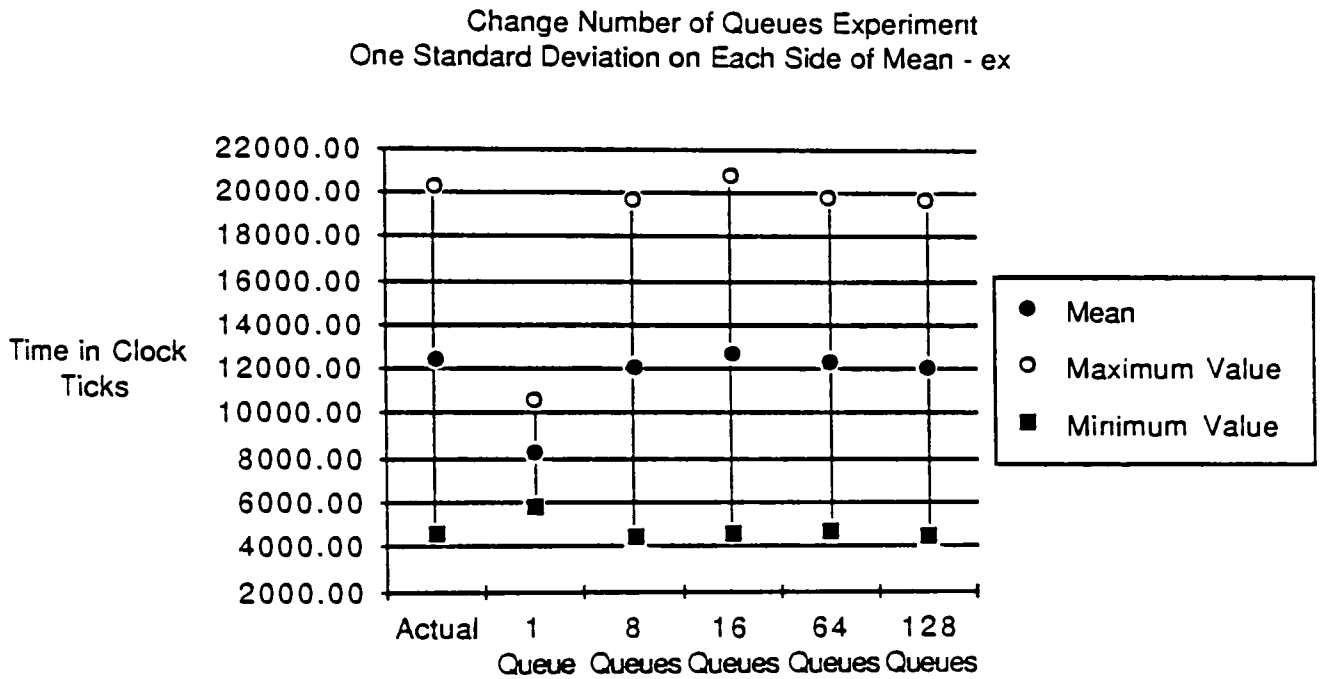

Change Number of Queues Experiment
One Standard Deviation on Each Side of Mean - ex
Time in Clock Ticks
22000.00
20000.00
18000.00
16000.00
14000.00
12000.00
10000.00
8000.00
6000.00
4000.00
2000.00
Actual
1 Queue
8 Queues
16 Queues
64 Queues
128 Queues
Mean
Maximum Value
Minimum Value

Chart sdcq5

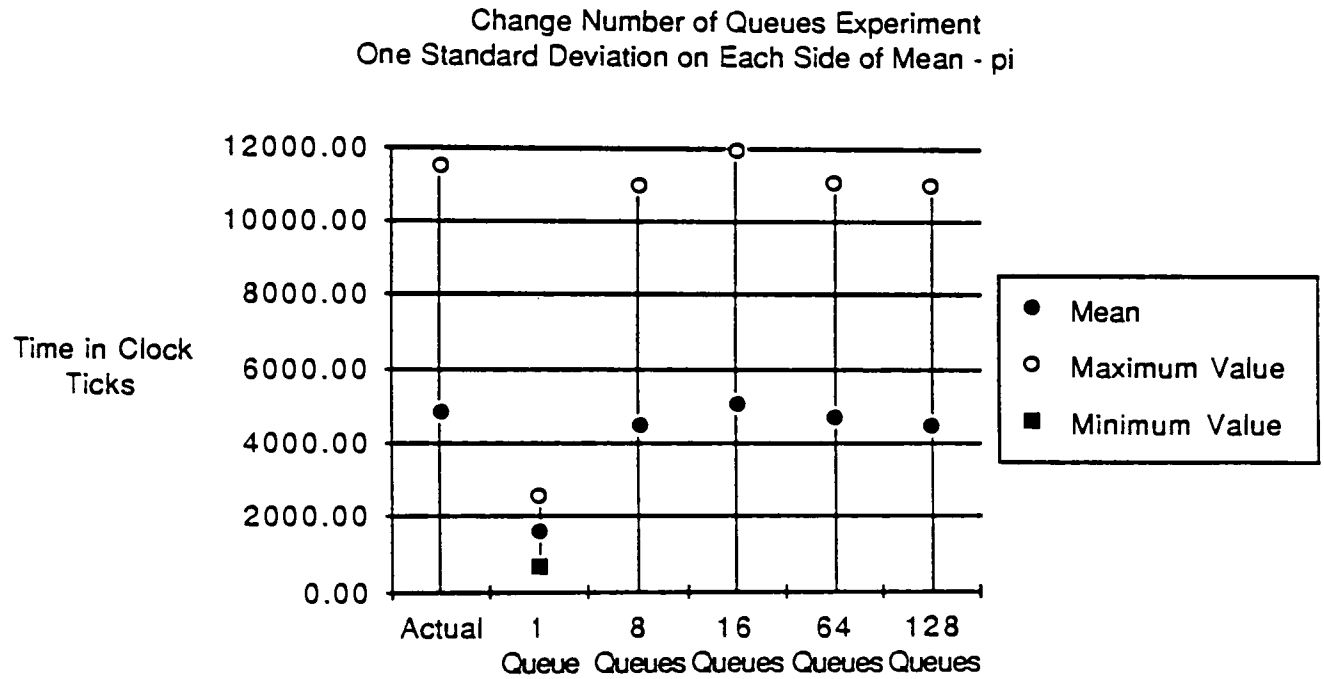

Change Number of Queues Experiment
One Standard Deviation on Each Side of Mean - pi
Time in Clock Ticks
12000.00
10000.00
8000.00
6000.00
4000.00
2000.00
0.00
Actual
1 Queue
8 Queues
16 Queues
64 Queues
128 Queues
Mean
Maximum Value
Minimum Value

Chart sdcq6

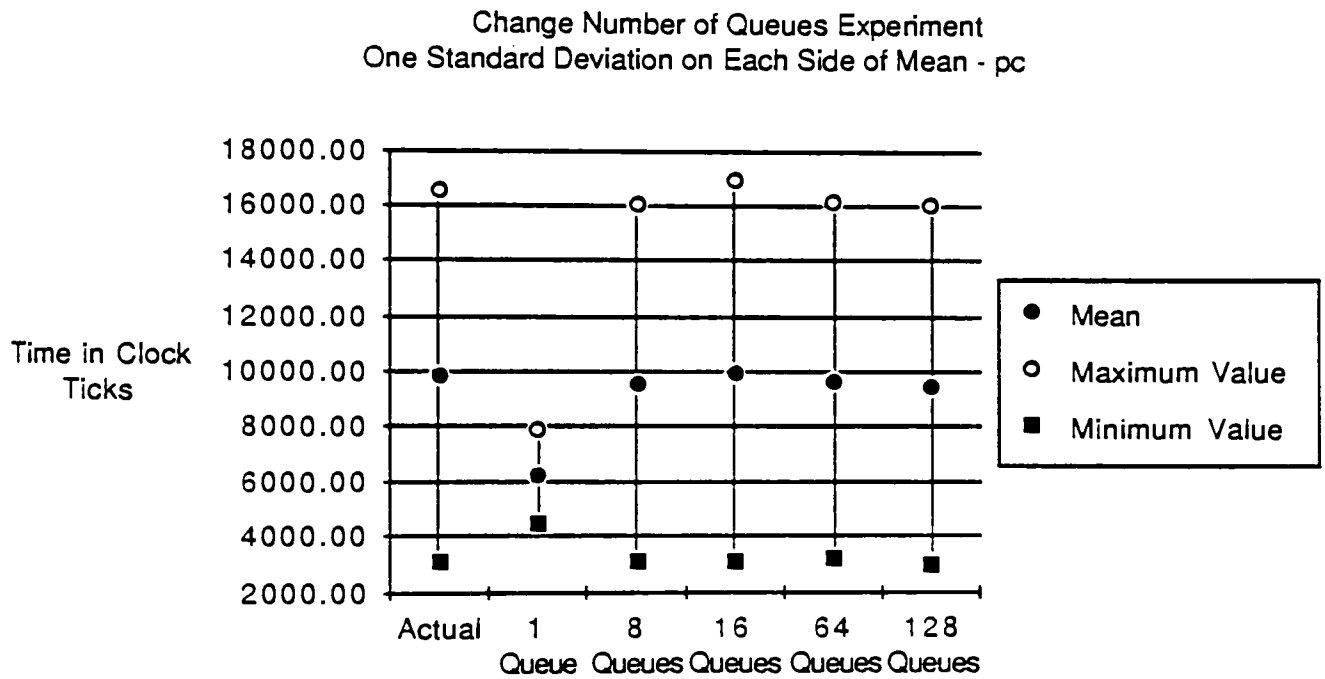
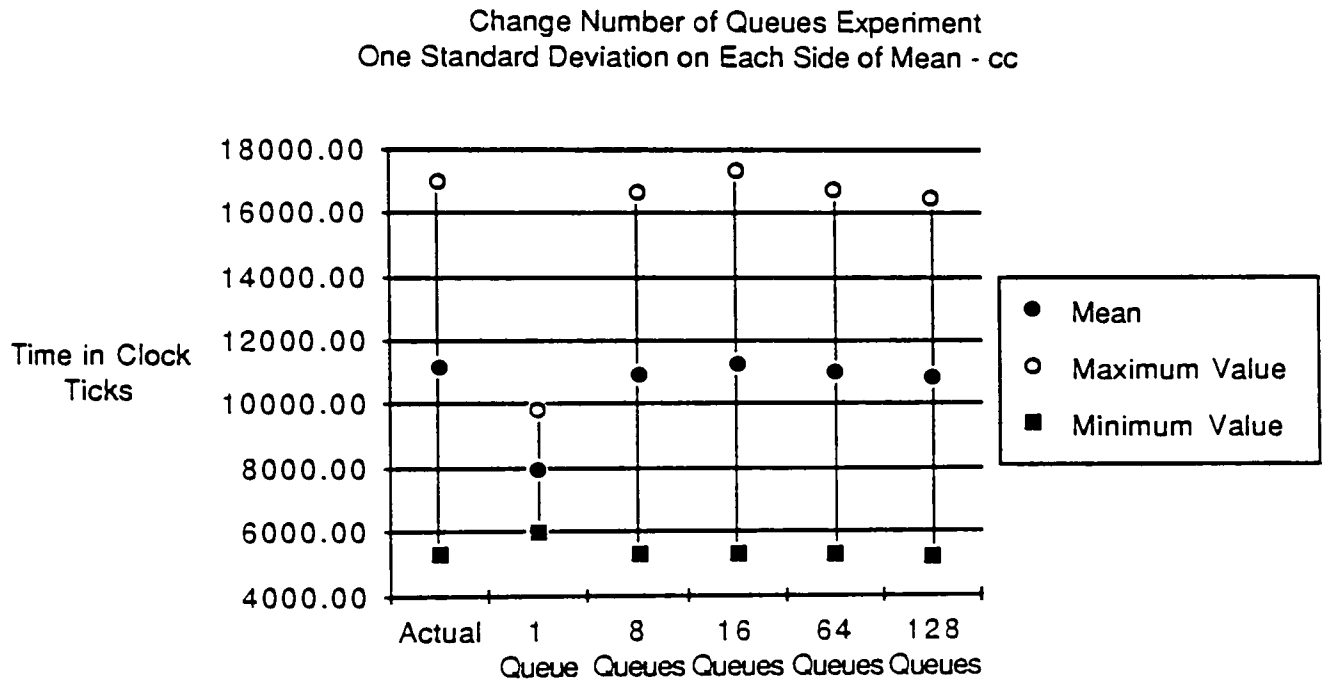

Change Number of Queues Experiment
One Standard Deviation on Each Side of Mean - pc
Time in Clock Ticks
18000.00
16000.00
14000.00
12000.00
10000.00
8000.00
6000.00
4000.00
2000.00
Actual Queue
1 Queues
8 Queues
16 Queues
64 Queues
128 Queues
Mean
Maximum Value
Minimum Value

Chart sdcq7


Change Number of Queues Experiment
One Standard Deviation on Each Side of Mean - cc
Time in Clock Ticks
18000.00
16000.00
14000.00
12000.00
10000.00
8000.00
6000.00
4000.00
Actual
1 Queue
8 Queues
16 Queues
64 Queues
128 Queues
Mean
Maximum Value
Minimum Value

Charts for the Delay Priority Drop Experiment

Introduction

The following charts show the results of the delay priority drop experiment. This experiment involved delaying the reduction of the priority of a process as it consumed CPU time. In the real system, the priority of a process is reduced every fourth tick consumed. In this experiment, fifty runs of the model were made in which the priority of a process was reduced every eighth tick, and another fifty runs were made in which the priority was reduced every sixteenth tick. Charts ses1 to ses7 show the 90% confidence intervals of the mean elapsed time for each member of the family of processes in both experiments. Charts sds1 to sds7 contain the intervals one standard deviation in width on each side of the mean elapsed time for each member of the family of processes. In both sets of charts, each chart includes the appropriate interval derived from runs of the unmodified model for comparison purposes.

Chart ses1

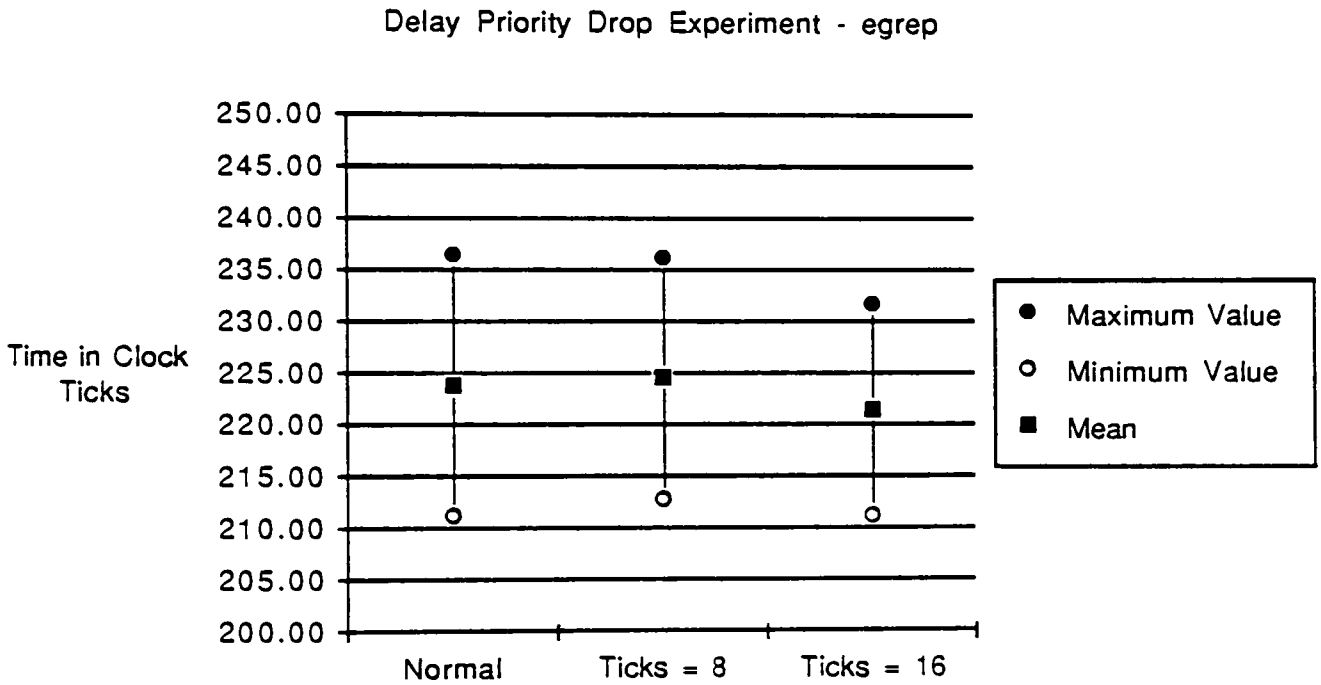

Delay Priority Drop Experiment - egrep
Time in Clock Ticks
250.00
245.00
240.00
235.00
230.00
225.00
220.00
215.00
210.00
205.00
200.00
Normal
Ticks = 8
Ticks = 16
Maximum Value
Minimum Value
Mean

Chart ses2

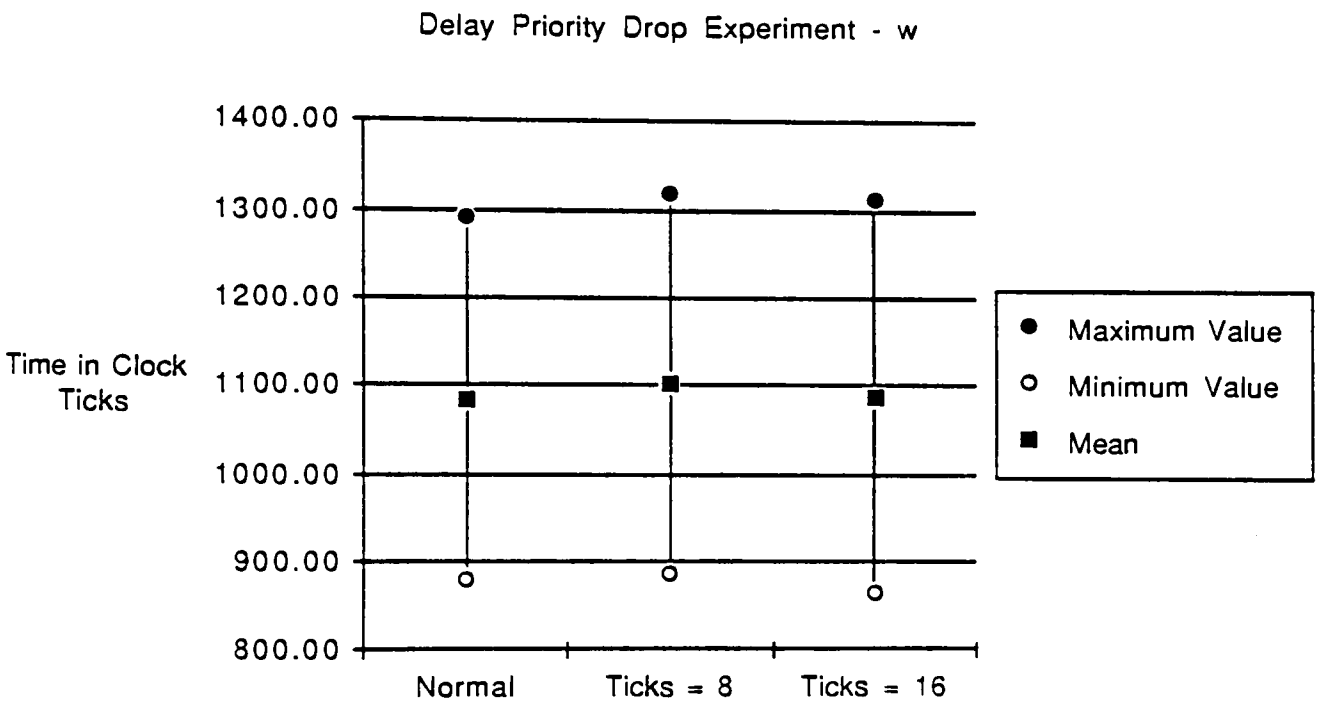

Delay Priority Drop Experiment - w
Time in Clock Ticks
1400.00
1300.00
1200.00
1100.00
1000.00
900.00
800.00
Normal
Ticks = 8
Ticks = 16
Maximum Value
Minimum Value
Mean

Chart ses3

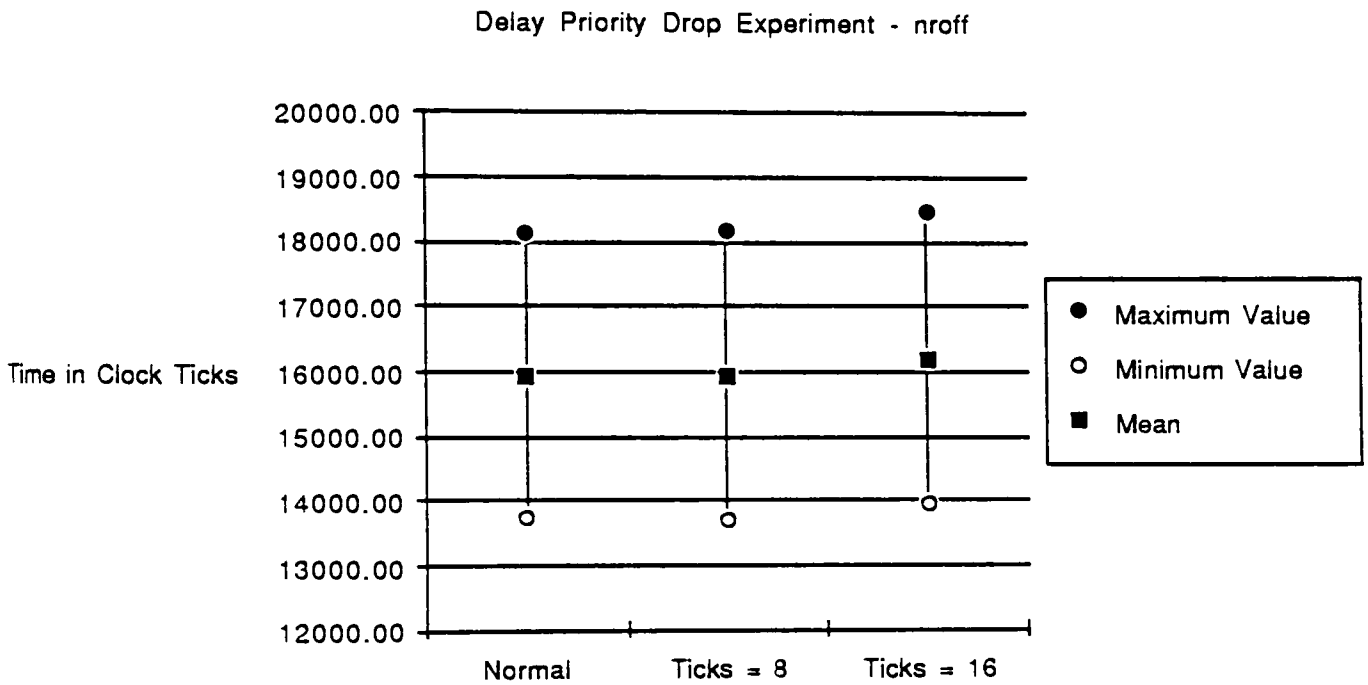

Delay Priority Drop Experiment - nroff
Time in Clock Ticks
20000.00
19000.00
18000.00
17000.00
16000.00
15000.00
14000.00
13000.00
12000.00
Normal
Ticks = 8
Ticks = 16
Maximum Value
Minimum Value
Mean

Chart ses4

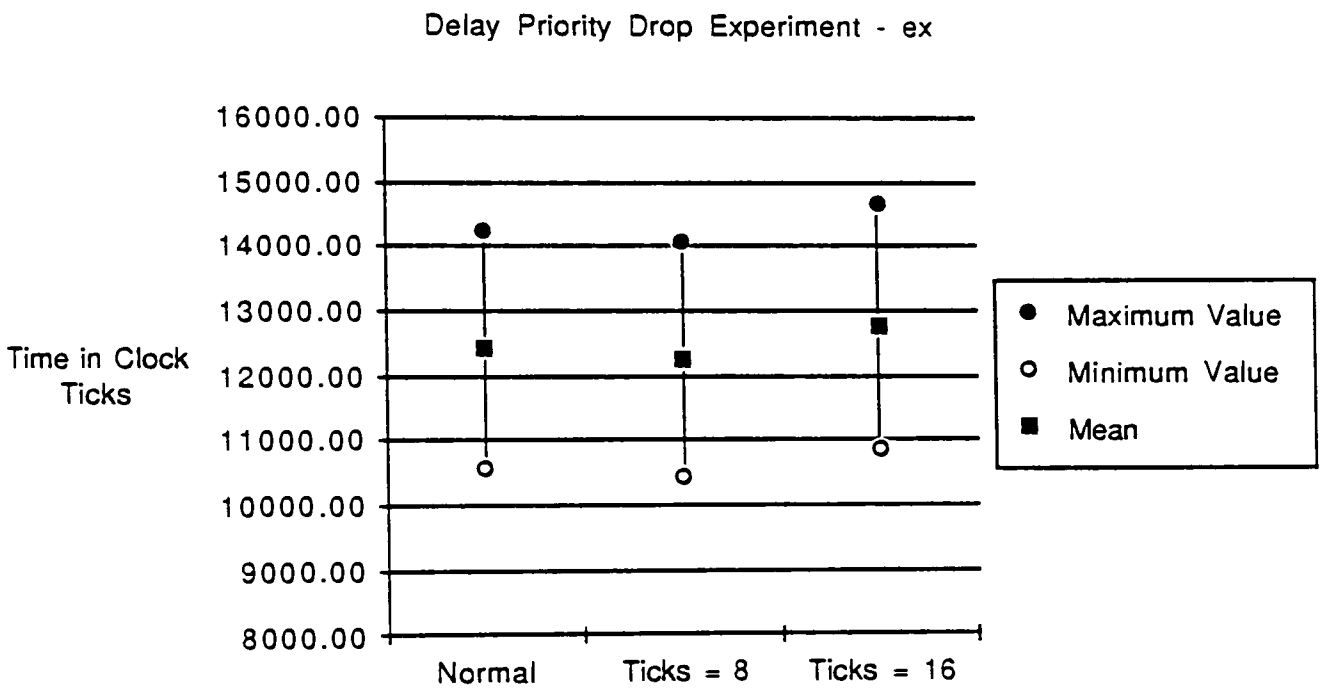

Delay Priority Drop Experiment - ex
Time in Clock Ticks
16000.00
15000.00
14000.00
13000.00
12000.00
11000.00
10000.00
9000.00
8000.00
Normal
Ticks = 8
Ticks = 16
Maximum Value
Minimum Value
Mean

Chart ses5

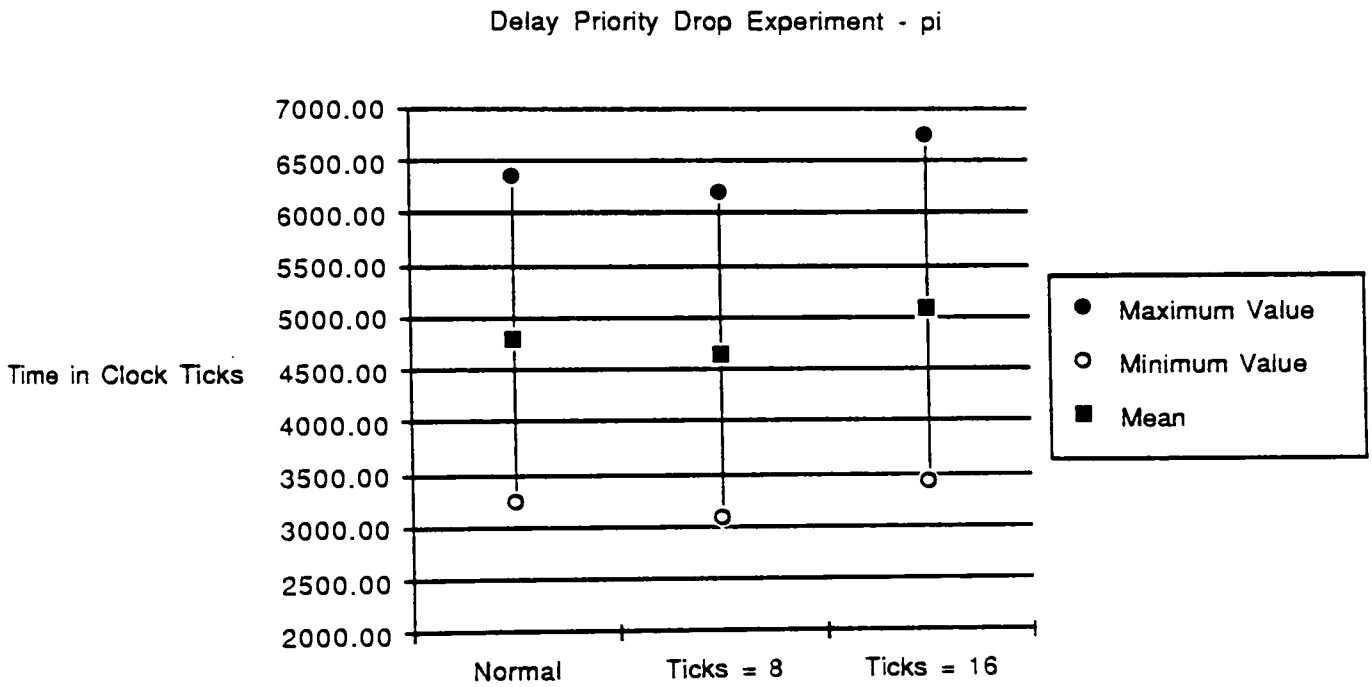

Delay Priority Drop Experiment - pi
Time in Clock Ticks
7000.00
6500.00
6000.00
5500.00
5000.00
4500.00
4000.00
3500.00
3000.00
2500.00
2000.00
Normal
Ticks = 8
Ticks = 16
Maximum Value
Minimum Value
Mean

Chart ses6

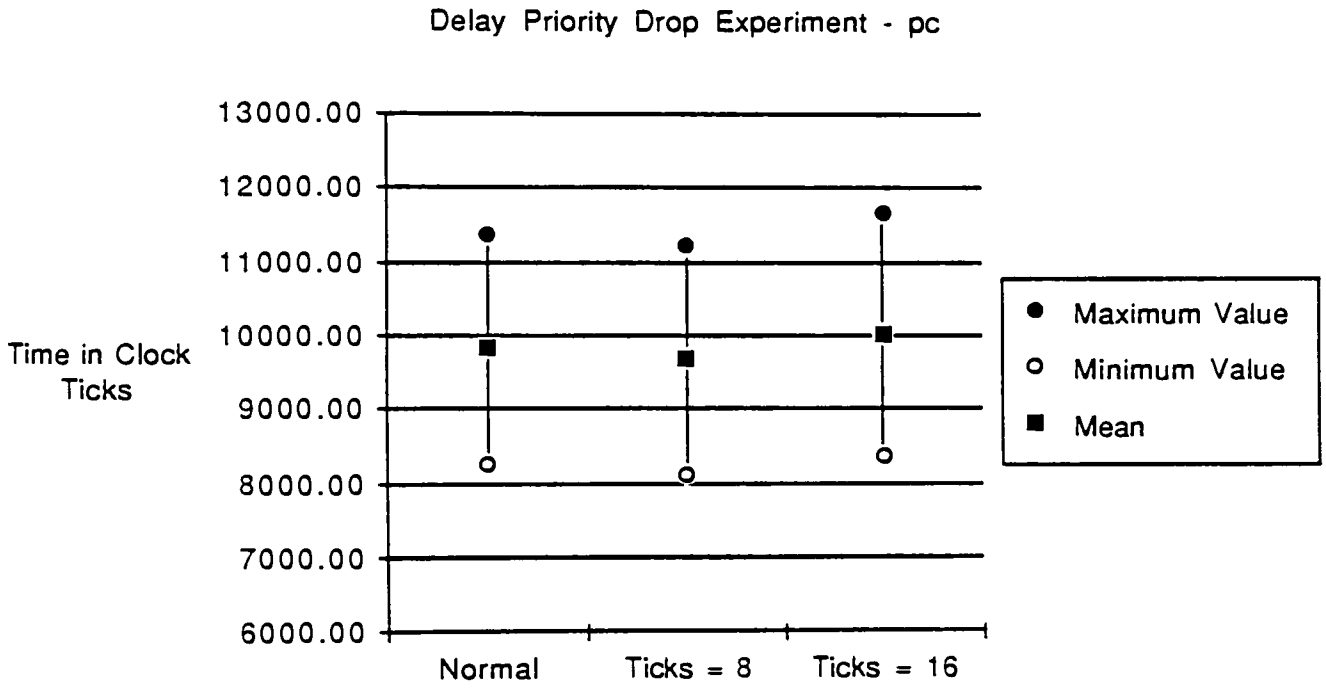

Delay Priority Drop Experiment - pc
Time in Clock Ticks
13000.00
12000.00
11000.00
10000.00
9000.00
8000.00
7000.00
6000.00
Normal
Ticks = 8
Ticks = 16
Maximum Value
Minimum Value
Mean

Chart ses7

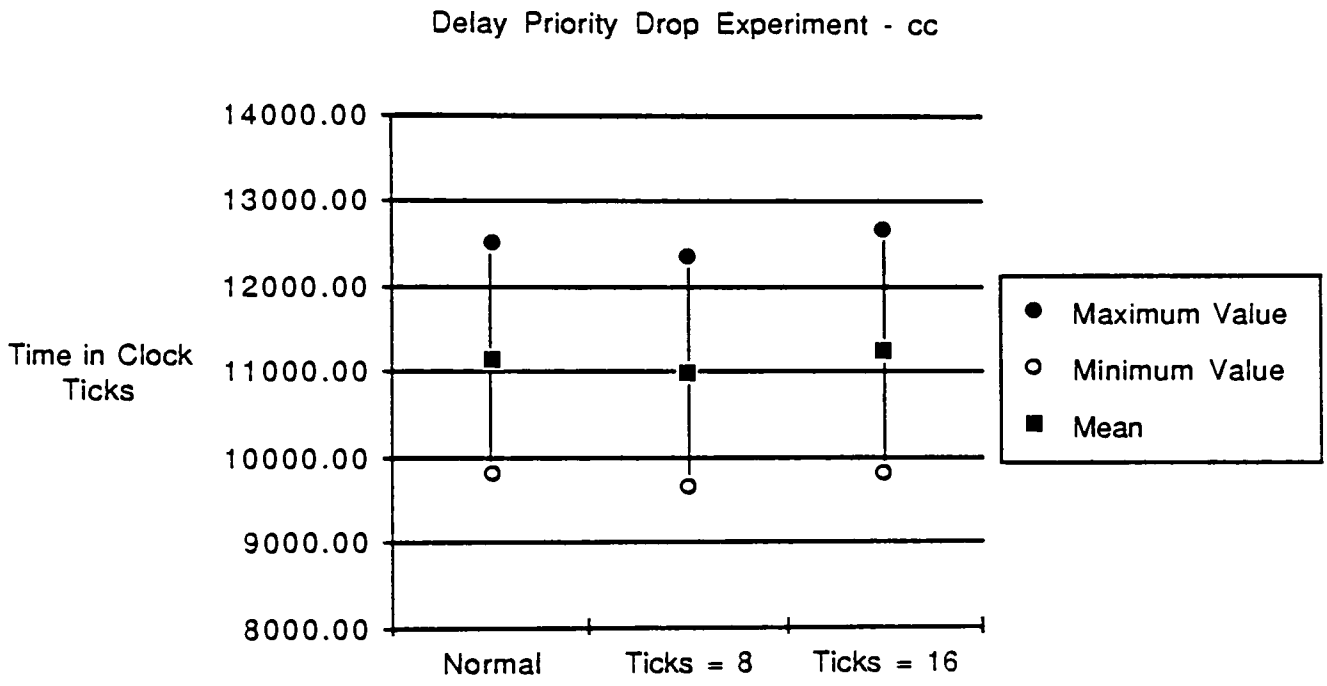

Delay Priority Drop Experiment - cc
Time in Clock Ticks
14000.00
13000.00
12000.00
11000.00
10000.00
9000.00
8000.00
Normal
Ticks = 8
Ticks = 16
Maximum Value
Minimum Value
Mean

Chart sds1

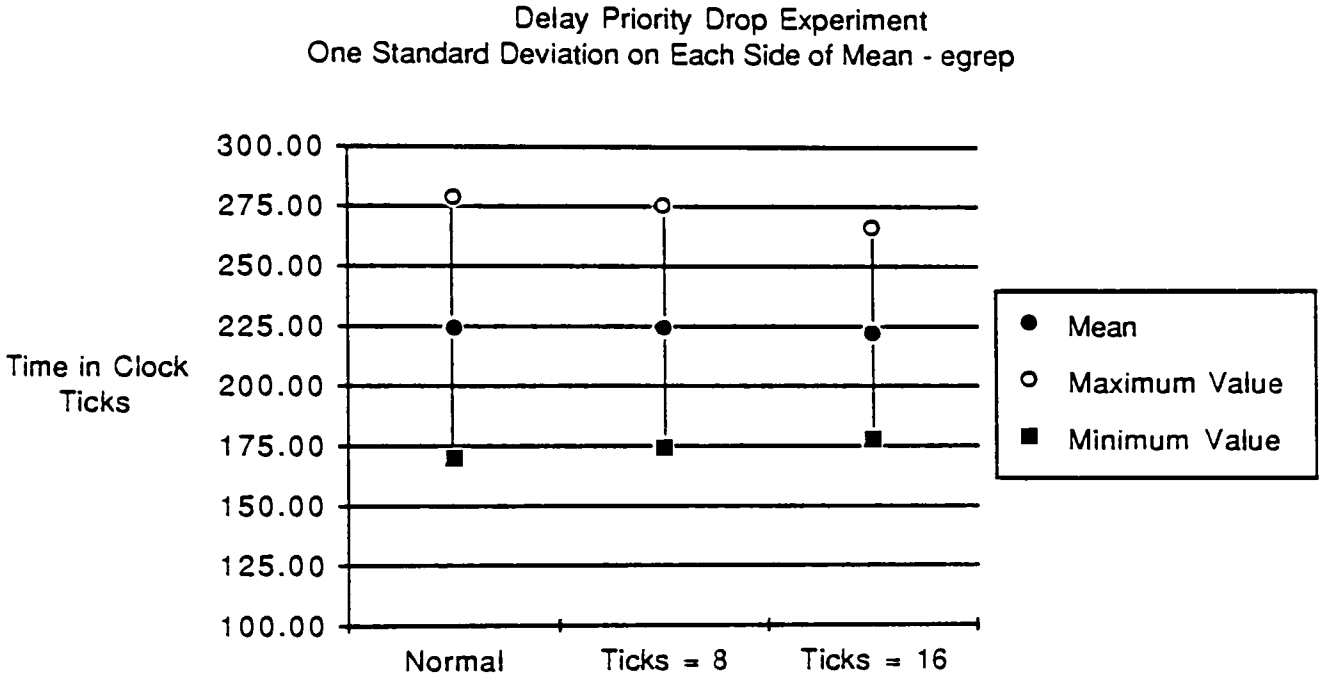

Delay Priority Drop Experiment
One Standard Deviation on Each Side of Mean - egrep
Time in Clock Ticks
300.00
275.00
250.00
225.00
200.00
175.00
150.00
125.00
100.00
Normal
Ticks = 8
Ticks = 16
Mean
Maximum Value
Minimum Value

Chart sds2

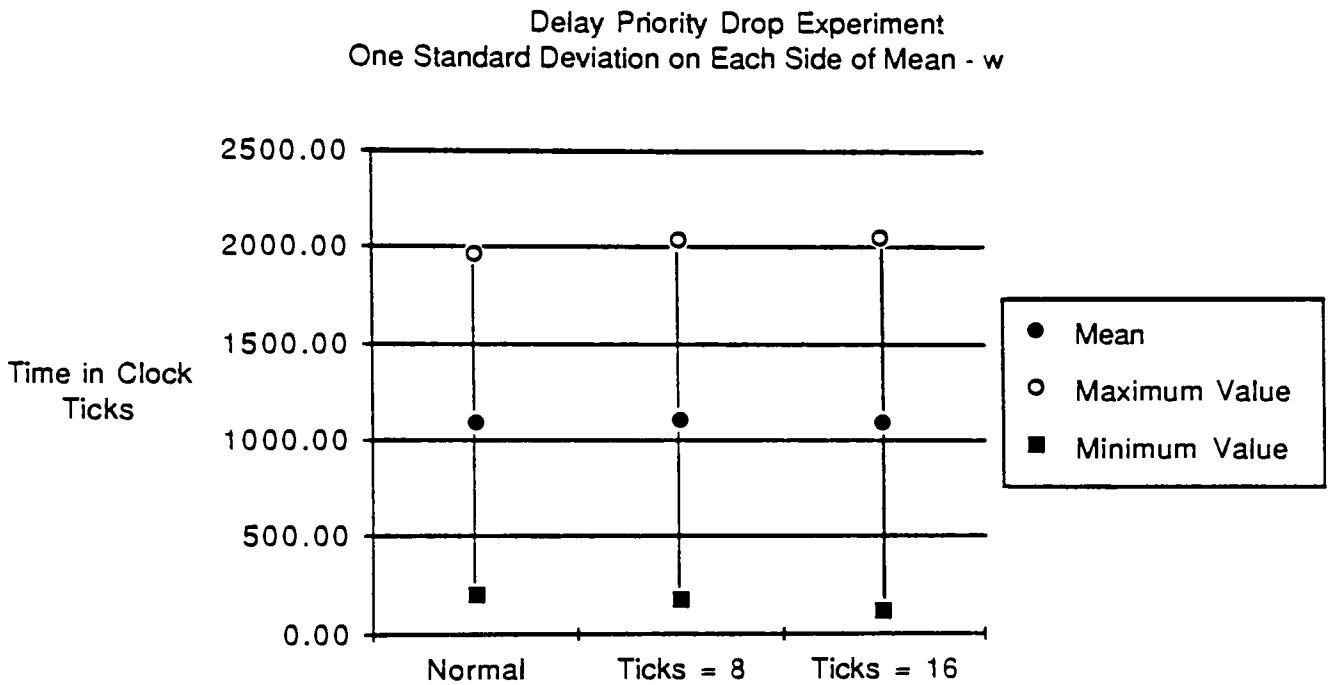

Delay Priority Drop Experiment
One Standard Deviation on Each Side of Mean - w
Time in Clock Ticks
2500.00
2000.00
1500.00
1000.00
500.00
0.00
Normal
Ticks = 8
Ticks = 16
Mean
Maximum Value
Minimum Value

Chart sds3

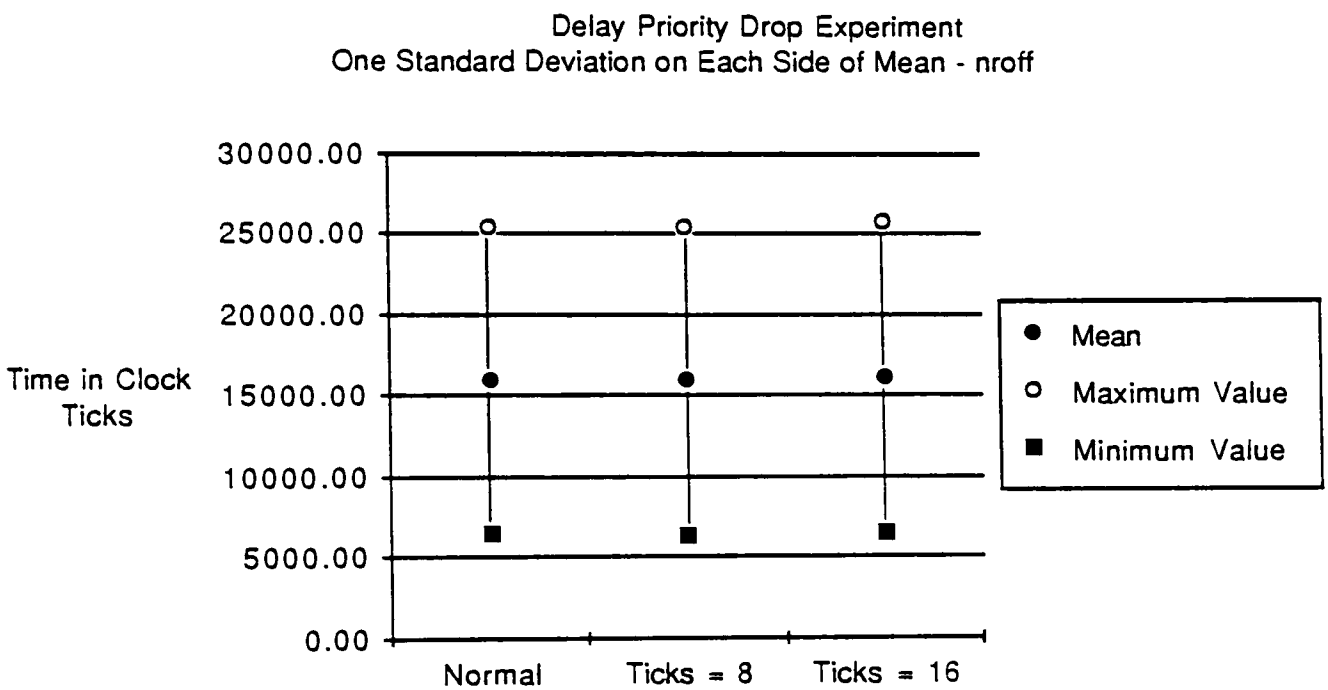

Delay Priority Drop Experiment
One Standard Deviation on Each Side of Mean - nroff
Time in Clock Ticks
30000.00
25000.00
20000.00
15000.00
10000.00
5000.00
0.00
Normal
Ticks = 8
Ticks = 16
Mean
Maximum Value
Minimum Value

Chart sds4

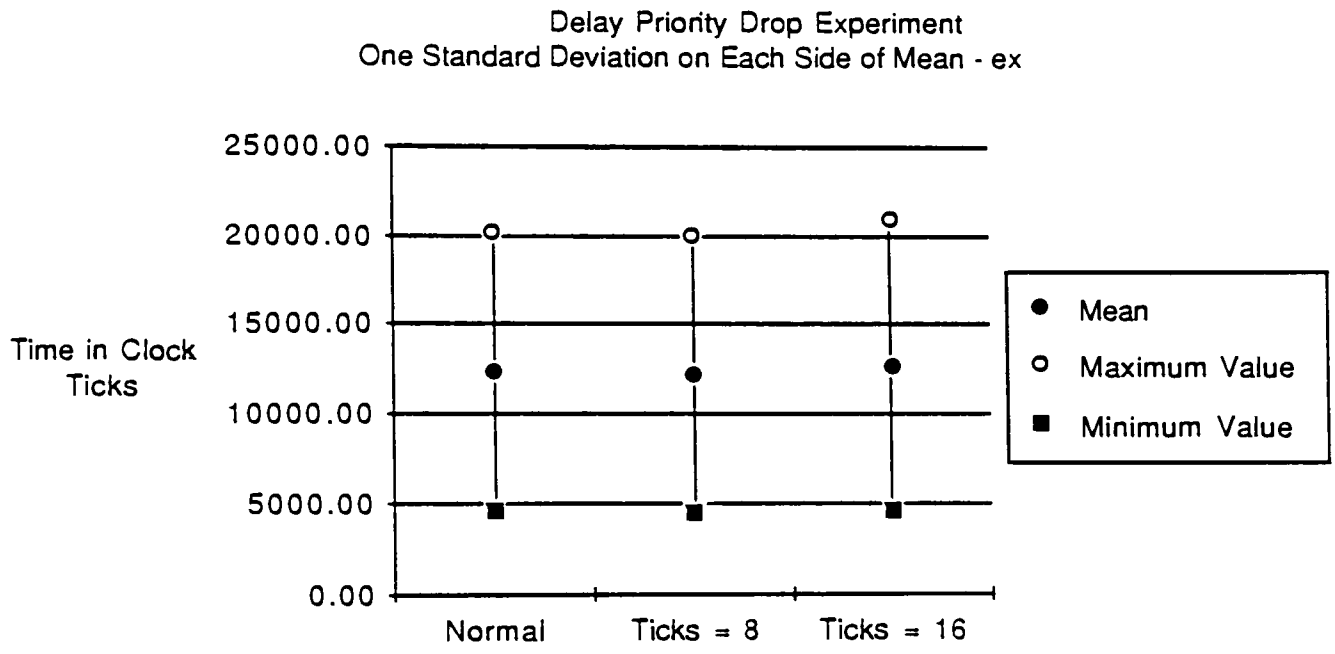

Delay Priority Drop Experiment
One Standard Deviation on Each Side of Mean - ex
Time in Clock Ticks
25000.00
20000.00
15000.00
10000.00
5000.00
0.00
Normal
Ticks = 8
Ticks = 16
Mean
Maximum Value
Minimum Value

Chart sds5

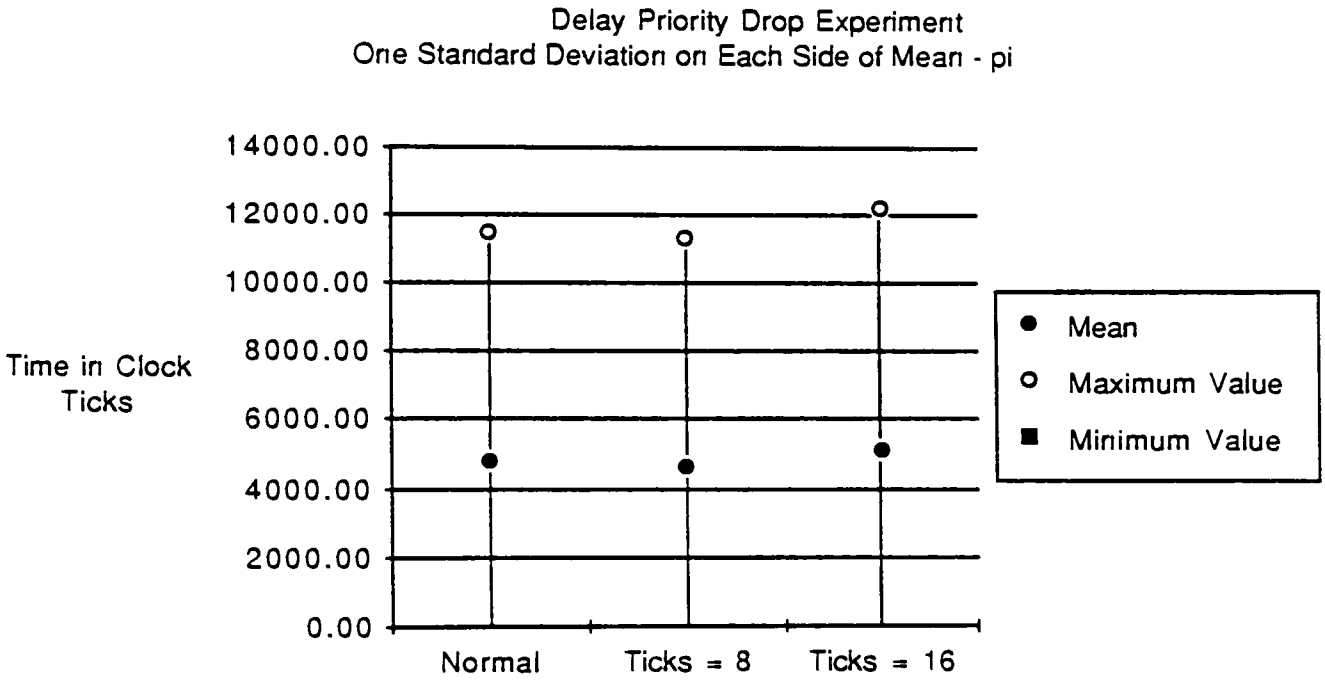

Delay Priority Drop Experiment
One Standard Deviation on Each Side of Mean - pi
Time in Clock Ticks
14000.00
12000.00
10000.00
8000.00
6000.00
4000.00
2000.00
0.00
Normal
Ticks = 8
Ticks = 16
Mean
Maximum Value
Minimum Value

Chart sds6

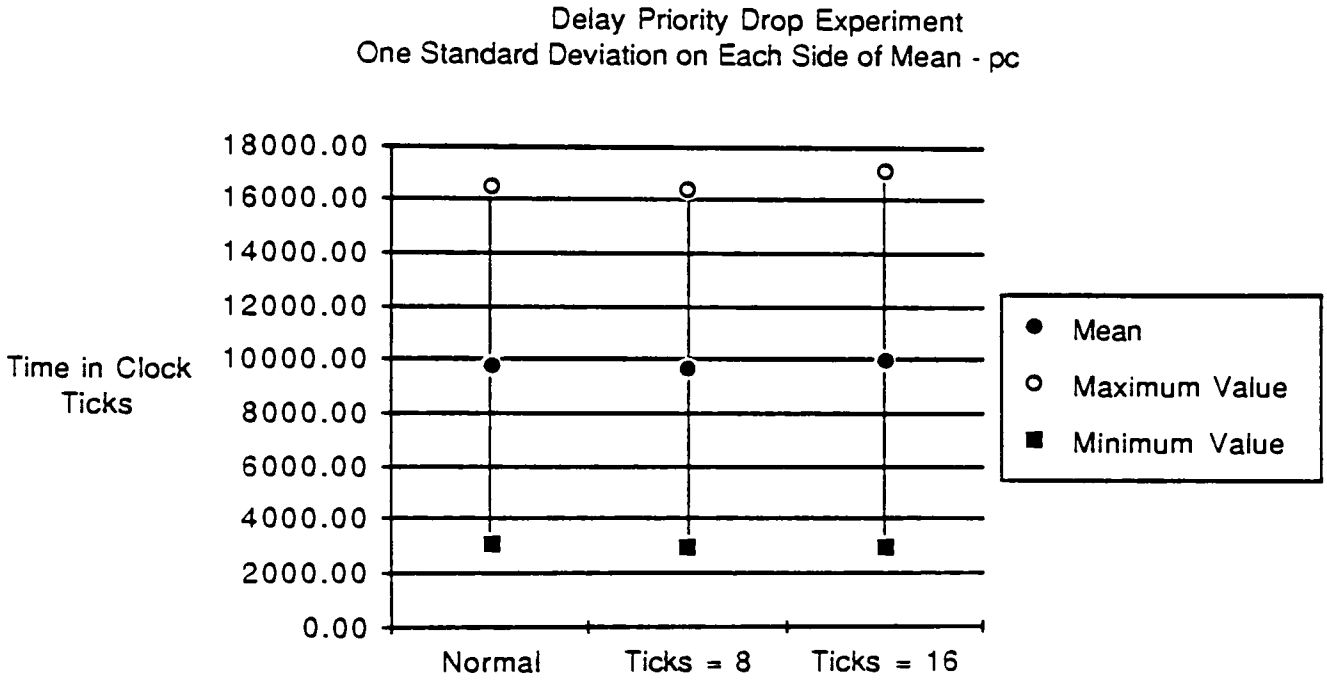
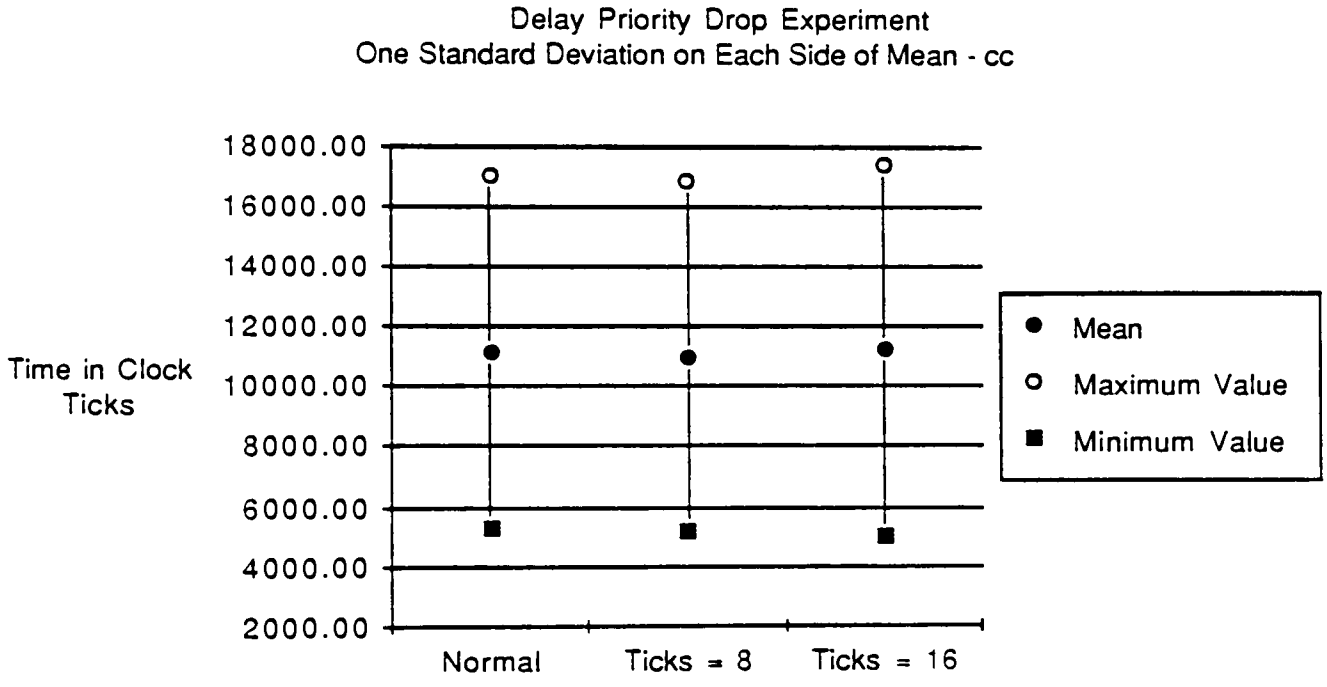

Delay Priority Drop Experiment
One Standard Deviation on Each Side of Mean - pc
Time in Clock Ticks
18000.00
16000.00
14000.00
12000.00
10000.00
8000.00
6000.00
4000.00
2000.00
0.00
Normal
Ticks = 8
Ticks = 16
Mean
Maximum Value
Minimum Value

Chart sds7


Delay Priority Drop Experiment
One Standard Deviation on Each Side of Mean - cc
Time in Clock Ticks
18000.00
16000.00
14000.00
12000.00
10000.00
8000.00
6000.00
4000.00
2000.00
Normal
Ticks = 8
Ticks = 16
Mean
Maximum Value
Minimum Value