July 1990

UILU-ENG-90-2224
CSG-126

COORDINATED SCIENCE LABORATORY
College of Engineering

DYNAMIC LOAD-SHARING USING PREDICTED PROCESS RESOURCE REQUIREMENTS

Kumar K. Goswami
Ravishankar K. Iyer

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-90-2224 CSG 126	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION NASA
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Ave. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) Langley Research Center Hampton, VA 23665
8a. NAME OF FUNDING / SPONSORING ORGANIZATION NASA 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NASA NAG 1-613
8c. ADDRESS (City, State, and ZIP Code) Langley Research Center Hampton, VA 23665	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
"Dynamic Load-Sharing using Predicted Process Resource Requirements"

12. PERSONAL AUTHOR(S)
Kumar K. Goswami and Ravishankar K. Iyer

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	1990, June 15	23

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	distributed systems, load sharing, statistical clustering, resource prediction, dynamic scheduling and trace-driven simulation

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

In the paper, heuristics which use *predicted* process resource requirements to make scheduling decisions are proposed. Four heuristics are presented. The first two, MINQ and SMPL, employ centralized scheduling and the remaining two, DMINQ and FDMINQ, use distributed scheduling. These heuristics are first compared against random scheduling and then against two conventional heuristics, CENTEX and DISTED, which schedule processes solely based on system state information. Results based on trace-driven simulations show that the proposed centralized heuristics offer significantly improved mean response times and, they require fewer status update messages. In experiments using the same status update rates, SMPL response times were, on the average, 22% lower than those for CENTEX and, MINQ response times were, on the average, 18% lower. The simulations also showed that MINQ and SMPL can perform as well as, or better than, CENTEX while using up to 70% fewer status update messages. The use of fewer status update messages imposes less overhead on the system. The use of prediction for distributed scheduling produced similar results. When prediction was used to filter small processes and execute them locally a 50% improvement in response times was obtained.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

Dynamic Load-Sharing using Predicted Process Resource Requirements

Kumar K. Goswami and Ravishankar K. Iyer
Center for Reliable and High Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 West Springfield Avenue
Urbana, Illinois 61801 USA

June 15, 1990

Abstract

In the paper, heuristics which use *predicted* process resource requirements to make scheduling decisions are proposed. Four heuristics are presented. The first two, MINQ and SMPL, employ centralized scheduling and the remaining two, DMINQ and FDMINQ, use distributed scheduling. These heuristics are first compared against random scheduling and then against two conventional heuristics, CENTEX and DISTED, which schedule processes solely based on system state information. Results based on trace-driven simulations show that the proposed centralized heuristics offer significantly improved mean response times and, they require fewer status update messages. In experiments using the same status update rates, SMPL response times were, on the average, 22% lower than those for CENTEX and, MINQ response times were, on the average, 18% lower. The simulations also showed that MINQ and SMPL can perform as well as, or better than, CENTEX while using up to 70% fewer status update messages. The use of fewer status update messages imposes less overhead on the system. The use of prediction for distributed scheduling produced similar results. When prediction was used to filter small processes and execute them locally a 50% improvement in response times was obtained.

Index Terms–Distributed systems, load sharing, statistical clustering, resource prediction, dynamic scheduling and trace-driven simulation.

1. Introduction

Dynamic load sharing heuristics have been studied extensively in the past. Most studies assume that the resource requirements of processes are not known *apriori* [Casavant 88]. However, better load sharing should be possible if the scheduler uses information on process resource requirements to make scheduling decisions. The results in [Devarakonda 89] show that it is possible to predict the CPU, memory, and I/O requirements of a process using a statistical pattern-recognition technique. This paper is concerned with the use of this prediction methodology to make scheduling decisions and to reduce the overhead of load sharing.

Four heuristics are proposed which use predicted process resource requirements to influence scheduling decisions. The first two (MINQ and SMPL) employ centralized scheduling and the remaining two (DMINQ and FDMINQ) use distributed scheduling. MINQ uses the predicted process resource requirements to determine the load (as measured by the CPU queue length) on the processors. The incoming processes are then directed to the processor with the least load. SMPL uses predicted process resource requirements to estimate the response time that a process will receive at each processor. The process is then sent to the processor offering the lowest estimated response time. DMINQ is simply a distributed version of the MINQ heuristic. FDMINQ employs a prediction-based filtering mechanism to identify and execute small processes locally.

The proposed heuristics which use prediction are first compared against random scheduling and then against two conventional heuristics, CENTEX and DISTED, which schedule processes based solely on system state information [Zhou 86]. CENTEX has been shown, through trace-driven simulations, to be a very effective centralized heuristic and DISTED is a distributed version of CENTEX.

Results based on trace-driven simulations show that the proposed centralized heuristics offer improved mean response times and perform well while using fewer status update messages. The simulations reveal that MINQ and SMPL perform as well as, or better than, CENTEX while using up to 70% fewer status update messages. In experiments using an equal number of status update messages, SMPL response times were 22% lower than those produced by CENTEX and, MINQ response times were 18% lower.

The use of prediction for distributed scheduling produced similar results. Under moderate to high loads, the response times for DMINQ were 18% lower than those of DISTED. When prediction was used for filtering small processes and executing them locally (FDMINQ) a significant reduction in response times was obtained; the mean response time was up to 50% lower than that produced by DISTED.

The following section discusses recent related work in this area and describes the technique used to predict process resource requirements. Section 3 presents the four proposed scheduling policies, MINQ, SMPL, DMINQ and FDMINQ. Sections 4 and 5 present the simulation model and the results of the trace-driven simulations, respectively. Finally, Section 6 summarizes the important findings and suggests directions for future research.

2. Background

The area of dynamic load balancing has been widely investigated ([Eager 86], [Hwang 82], [Krueger 84], [Leland 86], [Livny 82], [Stankovic 85], [Zhou86]). Studies that have a bearing with the research presented in this paper include [Livny 82] which presents a comprehensive study of several load-sharing heuristics and demonstrates that even with communication delays and processing overheads, load-sharing can improve the response time of a system. In [Stankovic 85] stability issues in load-sharing are discussed and heuristics that use a stochastic learning

automata to reduce instability are described. In [Barak 85] results of an actual implementation are described and the use of average rather than the instantaneous load measurements is discussed. In [Eager 86] an analytical study of three simple load-sharing policies is presented; the authors demonstrate that simple policies provide a significant improvement in response time. More complex policies are shown to provide only a marginal improvement over these simpler policies.

Research most closely related to that presented in this paper is that of [Zhou 86] and [Leland 86]. In [Zhou 86] a centralized, dynamic load-sharing policy, CENTEX, and a distributed policy, DISTED, are proposed. CENTEX uses the average CPU queue length to indicate the processor load. It directs incoming processes to the processor with the shortest queue length. Periodically, each processor sends an update message, consisting of its current CPU queue length, to the central scheduler. Using simulations, based on real trace data, the author shows that CENTEX produces lower overall mean response times than DISTED, a distributed version of CENTEX. Zhou also demonstrates that CENTEX performs as well, or better than, the three simple distributed policies described in [Eager 86].

Leland and Ott [Leland 86] analyzed 9.5 million Unix processes and found that the residual CPU time needed by a process is linearly related to its *age*,(i.e., The authors subsequently develop a *spiral assignment* scheme which schedules processes based on their age. Several heuristics based on this spiral assignment idea are developed and analyzed via trace-driven simulations.

The use of predicted process resource requirements to influence scheduling has not been fully explored in the literature. In systems using a round robin CPU scheduling policy, selection of the best processor depends on i) the number of processes in the processor, ii) the resource

requirements of these processes and, iii) the resource requirement of the process being scheduled. The first is easily obtained and the other two can be estimated by using a prediction technique. It can be argued that Leland and Ott implicitly use prediction since process age is used to estimate ("predict") its residual CPU requirement. The heuristics proposed in this study use a more direct approach.

The prediction method described in [Devarakonda 89] uses a statistical pattern-recognition-based approach to predict the CPU time, the file I/O and the memory usage of a process at the beginning of its life, given the identity of the program being executed[1]. The method was based on an analysis of over 65,000 UNIX processes. Initially, a statistical clustering algorithm is used to identify high-density regions of process resource usage. These regions (defined as states) are used to build a state-transition model to characterize the resource usage of the past executions of a program. The prediction scheme uses the resource usage of a program's last execution and the program's state-transition model to estimate the resource requirements for its next execution. In experiments using this approach, the coefficient of correlation between the predicted and actual CPU requirements of processes analyzed was found to be 0.84 out of 1.0. All the heuristics proposed in this study use Devarakonda's prediction scheme.

3.0 The Proposed Load Sharing Heuristics

This section proposes the four dynamic load-sharing heuristics that use predicted process resource requirements to make scheduling decisions. The first two (MINQ and SMPL) are centralized heuristics, and the remaining two (DMINQ and FDMINQ) are distributed versions of MINQ. All four heuristics assume that processes are logically independent of one another and

[1] A process is an execution or instance of a program.

are irrevocably assigned to a processor (i.e., once assigned a process is never migrated to another processor).

The framework for the centralized load-sharing heuristics is illustrated in Figure 1. The box containing the *predictor* and the *scheduler* is referred to as the central scheduler. When a process arrives at the central scheduler, its identification is sent to the predictor which predicts the resource requirements of the process. These predicted values are then fed to the scheduler which, based on the specifics of the load-sharing policy, identifies the processor that will house the process. When a process completes execution, its *actual* resource usage is stored by the processor in a buffer. Periodically, each processor sends a status update message, consisting of the contents of this buffer, to the central scheduler which in turn uses this information to update the appropriate state transition model.

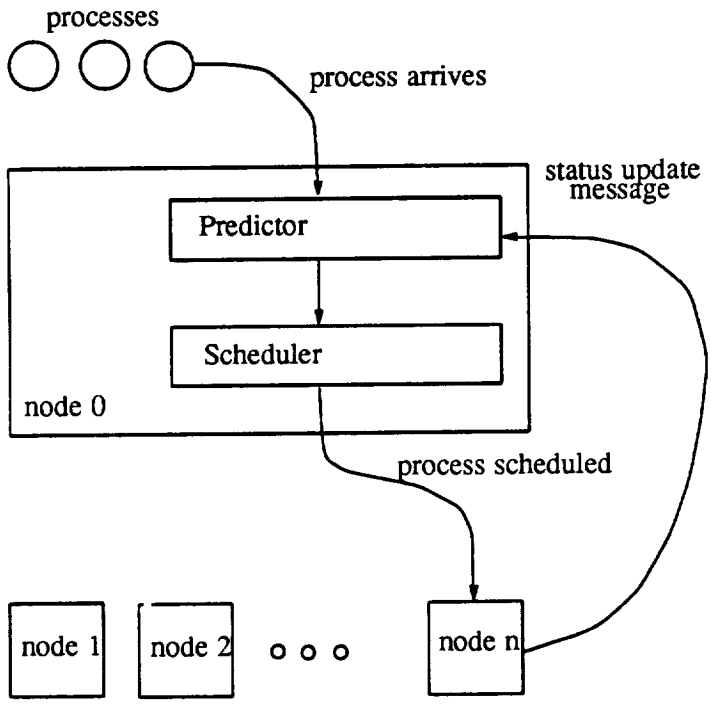

Figure 1. Framework of centralized load-sharing with prediction

Figure 2 depicts the framework for the distributed heuristics. The approach is similar to centralized scheduling, except that now each node has its own scheduler. When a process arrives at a node it is first sent to the predictor to estimate its resource requirements. Next, the scheduler is invoked to select the processor that will house the process. Periodically each processor sends a status update message to all the other processors.

3.1 The MINQ Load-Sharing Heuristic

The MINQ scheduling policy is the simplest of the two centralized heuristics. For each processor, i, the scheduler maintains a queue containing the predicted CPU and I/O requirements of every process executing in the processor. When a process, X, arrives, the scheduler estimates the CPU load on each processor and sends the process to the processor with the

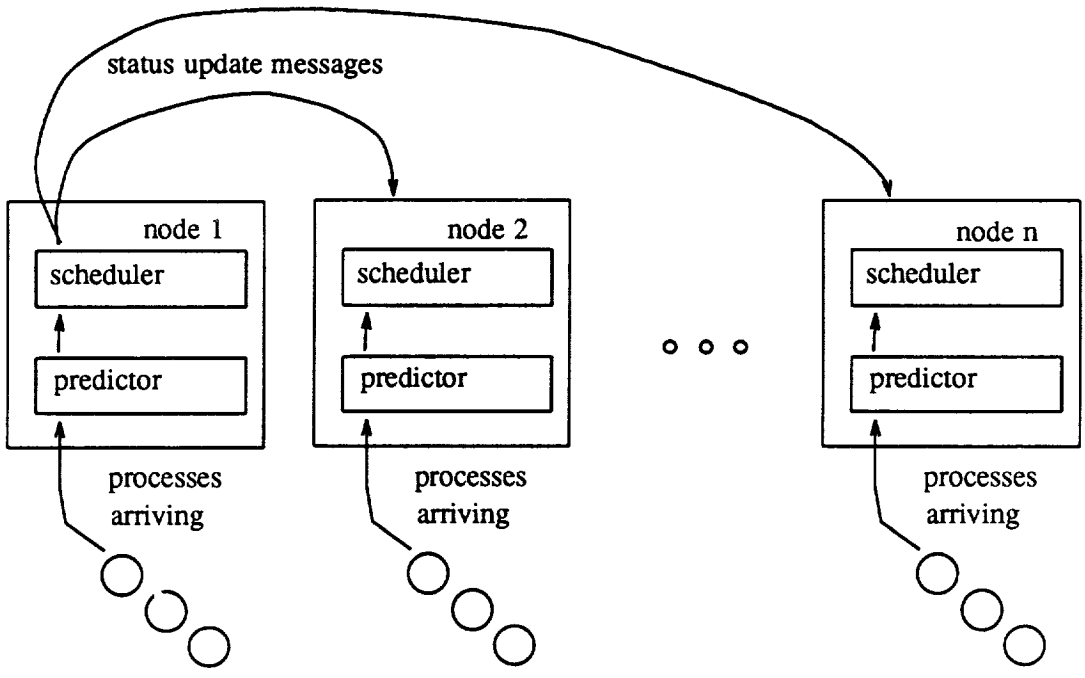

Figure 2. Framework of distributed load-sharing with prediction

lowest estimated load. The sequence of steps taken by the scheduler is outlined below.

1. The estimated average CPU load at each processor i is computed as follows:

$$CPU\_LOAD_i = \sum_{j=1}^{N_i} \frac{CPUREQ_j}{CPUREQ_j + IOREQ_j}$$

where:
N_i = the number of processes in processor i
$IOREQ_j$ is the predicted I/O requirement of process j in units of time
$CPUREQ_j$ is the predicted CPU requirement of process j in units of time

Note that *CPU_LOAD* is an estimate of the CPU queue length.

2. The process identification is fed to the predictor to obtain the predicted CPU and I/O requirement.
3. The process is sent to processor k with the smallest *CPU_LOAD* value.
4. An entry containing the predicted CPU and I/O requirement of process X is added to processor k's queue.

Recall that at periodic intervals each processor sends a status update message consisting of the actual resources used by its processes to the central scheduler. Upon receiving a status update message, the central scheduler performs the following processing steps for each process in the message.

1. The process identification and the actual CPU and I/O usage figures are fed to the predictor in order to update the state transition diagram for that program.
2. The appropriate process entry is deleted from the queue of the processor which sent the message.

3.2 The SMPL Load-Sharing Heuristic

When scheduling a process, SMPL takes the number of processes in a processor, the CPU requirement of these processes and the CPU requirement of the process being scheduled, into

account. Like the MINQ heuristic, SMPL maintains a queue for each processor. As processes arrive and are scheduled, entries are placed in the appropriate queue and as processes are completed, their entries are deleted from the queues. The fundamental difference between MINQ and SMPL lies in the policy used to select the processor with the least load. Recall that MINQ simply picks the processor with the smallest estimated CPU load. However, when round robin CPU scheduling is used, selecting the processor with the smallest CPU load may not necessarily guarantee the best response time for the process. When a process, X, is to be scheduled, the SMPL approach estimates the response time that the process will receive at each processor and selects the processor offering the lowest estimated response time. For each processor, i, the scheduler computes the response time in two steps. First, it sums the predicted CPU requirement of all processes smaller than process X. Next, for each of the remaining processes, an amount equal to the CPU requirement of process X is added to this sum. This final value is the estimated response time that X will receive at processor i. The detailed steps performed by SMPL are outlined below.

1. The predicted resource requirements of process X ($CPU\_REQ_{processX}$, $IO\_REQ_{processX}$) are determined via the prediction mechanism.
2. Given a round robin CPU scheduling discipline, the estimated response time, r_i, that process X will receive at each processor, i, is computed:

$$r_i = \sum_{j=1}^{N} I \times CPU\_REQ_j + (1-I) \times CPU\_REQ_{processX}$$

$$I = \begin{cases} 1 & \text{if } CPU\_REQ_j < CPU\_REQ_{processX} \\ 0 & otherwise \end{cases}$$

where:
N_i = the number of processes in processor i
CPU_REQ_j = the amount of CPU required by process j

3. The processor with the lowest r_i value is selected to house process X.
4. Process X is added to the appropriate processor's queue.

As in the MINQ heuristic, the central scheduler periodically receives status update messages containing the actual I/O and CPU time used by the processes that have been completed. The SMPL load-sharing heuristic performs the same status update processing as MINQ.

3.3 The DMINQ Load-Sharing Heuristic

In DMINQ (a distributed version of MINQ) each processor has a local scheduler which maintains a separate queue for each processor in the system. Queue i contains the predicted CPU and I/O requirements of every process that the local scheduler has sent to processor i. When a process arrives at a processor, if the load on the processor is less than a pre-specified threshold T, the process is housed in the same processor[2]. Otherwise, the local scheduler estimates the CPU load on every processor in the system via the information in its queues, and assigns the process to what it believes to be the least loaded processor. The scheduler then adds the process to its queue for the selected processor.

Periodically, each processor broadcasts a status update message, containing a list of all the processes that have been completed since the last status message was broadcast. The local schedulers use this message to refresh their global view of the system.

3.4 The FDMINQ Load-Sharing Heuristic

FDMINQ is identical to DMINQ except that, instead of using a fixed threshold mechanism based on the number of processes in a processor, it uses a filtering mechanism based on prediction. Predicted resource requirements are used to identify and filter out small processes (i.e.,

[2] A similar threshold mechanism is used by DISTED [Zhou 86]. In addition, DISTED also filters processes based on their name. Studies by Zhou show that certain processes are typically large while others are usually small and that process names can be used to distinguish between them. Due to a lack of implementation detail in [Zhou 86], this feature was not implemented for the DISTED heuristic used in this paper.

processes requiring little CPU time). Thus, regardless of the load on the processor, all small processes are executed locally. As will be seen, this reduces the burden on the scheduler and significantly improves the response times for small processes.

4.0 Experiment Design

4.1 The System

The load-sharing heuristics were tested on a simulated distributed system consisting of homogeneous processors that are connected by a single communication channel. Each processor is assumed to have infinite memory and use a pre-emptive round robin CPU scheduling discipline with a 100 millisecond time-slice. Process scheduling and message transfer have priority over process execution. A distributed file system is assumed so that the cost of accessing files is roughly the same for all hosts. This model is representative of a typical Ethernet-based distributed environment[3].

Only the CPU overhead of sending and receiving status update messages by the load-sharing heuristic is modeled. Thus, the I/O overhead and the message traffic produced by the application processes and by the load-sharing heuristics are not modeled. This assumption gives us conservative results. As indicated by the measurements in section 5.1, consideration of the impact of message traffic will only further enhance our results. Twenty milliseconds of CPU time is assumed to be needed to send a status message and 10 milliseconds is needed to receive and process a status update message. A cost of 100 milliseconds of CPU time is incurred by a processor when a process is transferred to it. These estimates, for the type of system studied,

[3] It should be emphasized that the proposed heuristics can be adapted to the specifics of a given topology.

were obtained from [Zhou 86]. Our measurements show that it takes approximately 5 milliseconds of CPU time to make a prediction. This cost was also included in the simulations.

4.2 Input Trace File

An actual trace file of 37,000 processes, executed on a VAX 11/780 running 4.3BSD Unix, was used as input to the simulated system. Each process in the trace file was defined by its identification number, its arrival time, its CPU, I/O, and memory requirements. Since the trace file contains logical I/O performed by a process, a file cache with a hit ratio of 75% was modeled. A file cache access time of 0.2 milliseconds and a cache miss time of 70 milliseconds was assumed. The process arrival rate was varied to observe the system under various loads. In simulating the distributed heuristics, processes were read from the trace file and randomly sent to the processors. Fifteen thousand processes were input to the system for each experiment.

5.0 Experiment Results and Discussion

Experiments were conducted to investigate system sizes varying from 6 to 25 nodes. Since the basic findings were similar, only results

Table 1. Process Arrival Rates for each Load

Load	Arrival Rate (processes/sec)
1	2.8
2	3.6
3	4.7
4	5.7
5	7.1
6	8.1

of experiments conducted on a 20 node system are presented in this paper. The performance metric used to judge the load-sharing policies was the mean response time of the processes. All

the heuristics were simulated using various parameters (e.g., status update interval, threshold value) and only the best response times attained are shown in Table 1.

The range of loads under which the heuristics were executed are also shown in Table 1.

5.1 Comparison of Centralized Heuristics

5.1.1 Comparison of MINQ, SMPL and CENTEX

Typically, random assignment is used as a reference against which new heuristics are compared. Figure 3 shows a comparison of MINQ against RND which randomly assigns a process to a processor. It is clear that MINQ yields substantial performance gains over RND at all loads tested.

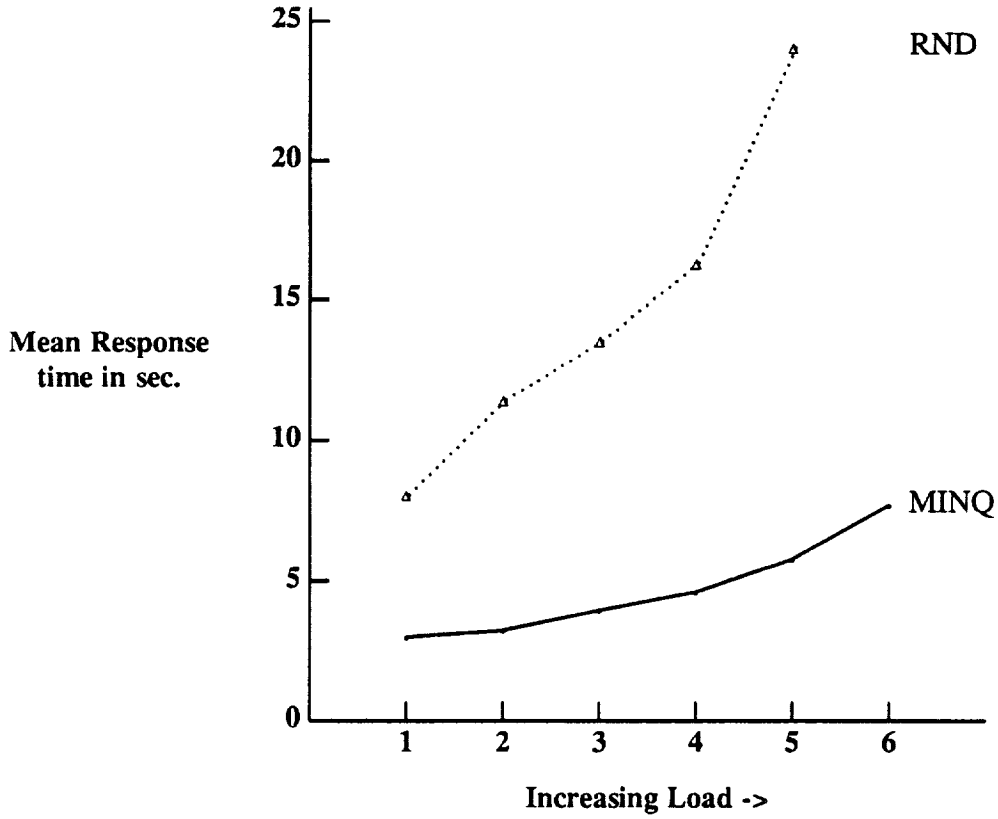

Figure 3. Comparison of MINQ and a random policy

Figure 4 compares the best response times obtained by MINQ and SMPL, which use prediction, and CENTEX, which does not use prediction. MINQ and SMPL consistently yield better response times, for all range of loads, than those produced by CENTEX. The response times of SMPL are as much as 30% lower than CENTEX. On the average, SMPL response times are 21% lower than CENTEX and MINQ response times are about 18% lower.

MINQ and SMPL perform better than CENTEX because they use predicted process resource requirement information to maintain an accurate running estimate of the processor load. When a process is to be scheduled, its predicted resource requirements are used to determine the load that it will place upon a processor. CENTEX, on the other hand, has no process specific

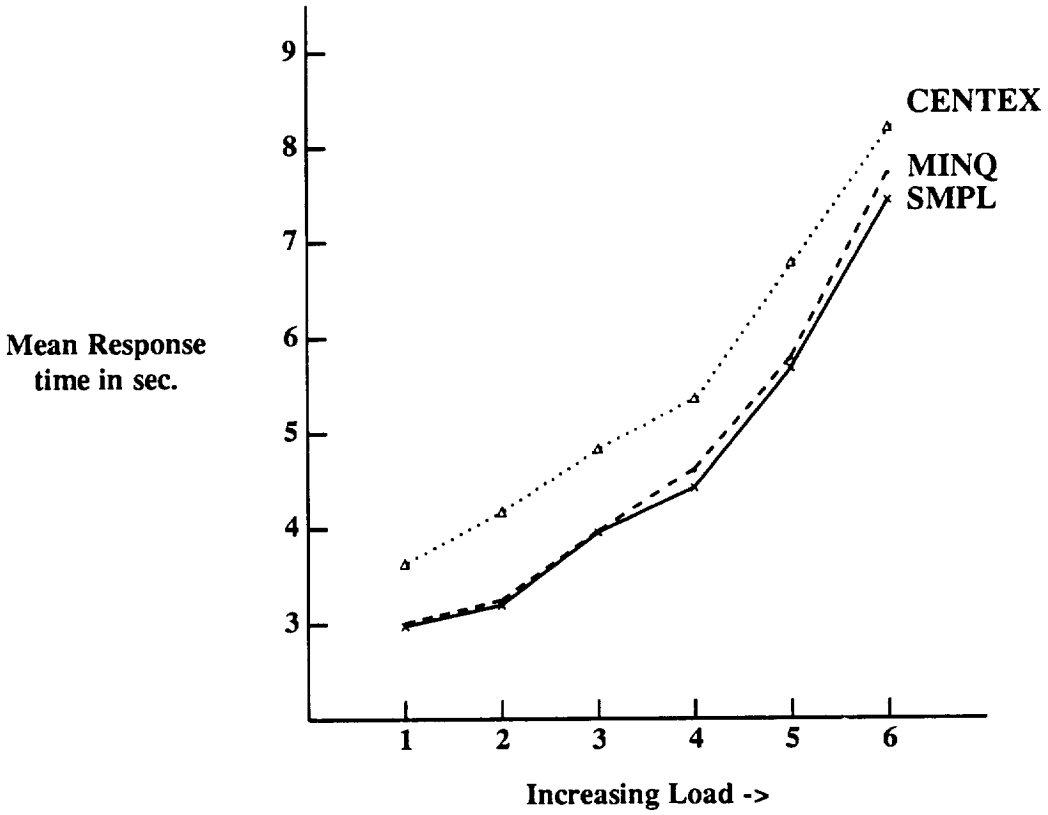

Figure 4. Comparison of MINQ, SMPL & CENTEX

information and hence has no *a priori* knowledge of the load imposed by a process. In order to estimate the load, CENTEX keeps a running account of the CPU queue length at each processor and, adds a constant (typically a 1) to the processor's queue length each time a process is sent to it.

Figure 4 also shows that SMPL performs only slightly better than MINQ. The advantage of SMPL, as shown in the next subsection, is that its performance is less sensitive to the status update interval than that of MINQ.

5.1.2 Impact of Varying Status Update Intervals

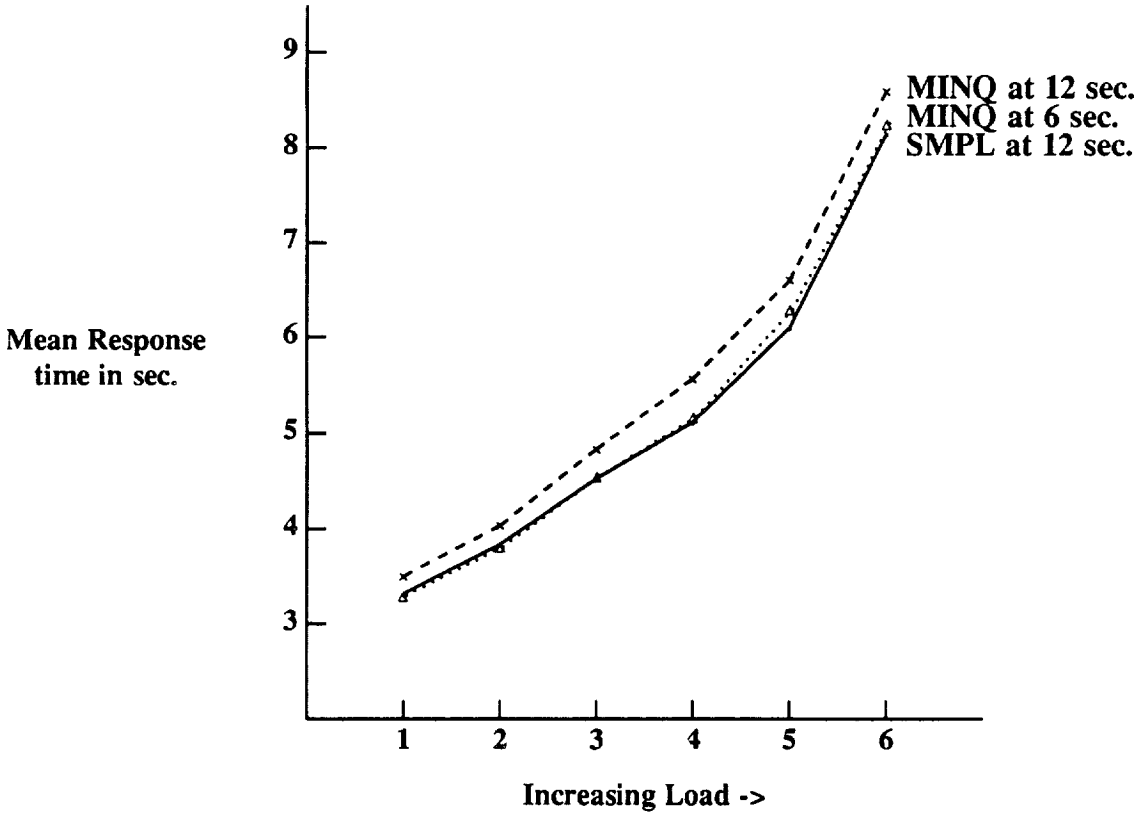

Figure 5a. MINQ and SMPL using various status update rates

A load-sharing heuristic which requires frequent status updates can significantly increase the CPU time needed for message processing. In addition, it can substantially contribute to the message traffic in the system. Prediction makes it possible to lower these overheads because it allows MINQ and SMPL to maintain their performance while using fewer status update rates.

Figure 5a compares SMPL and MINQ with respect to status update periods. The figure shows the response times for SMPL with a 12 second update interval and compares it with MINQ with 6 and 12 second update intervals. It is clear that SMPL performs just as well as MINQ while using a status update interval that's twice as slow. The reason is because SMPL explicitly tries to predict the response time as opposed to MINQ, which simply predicts the load

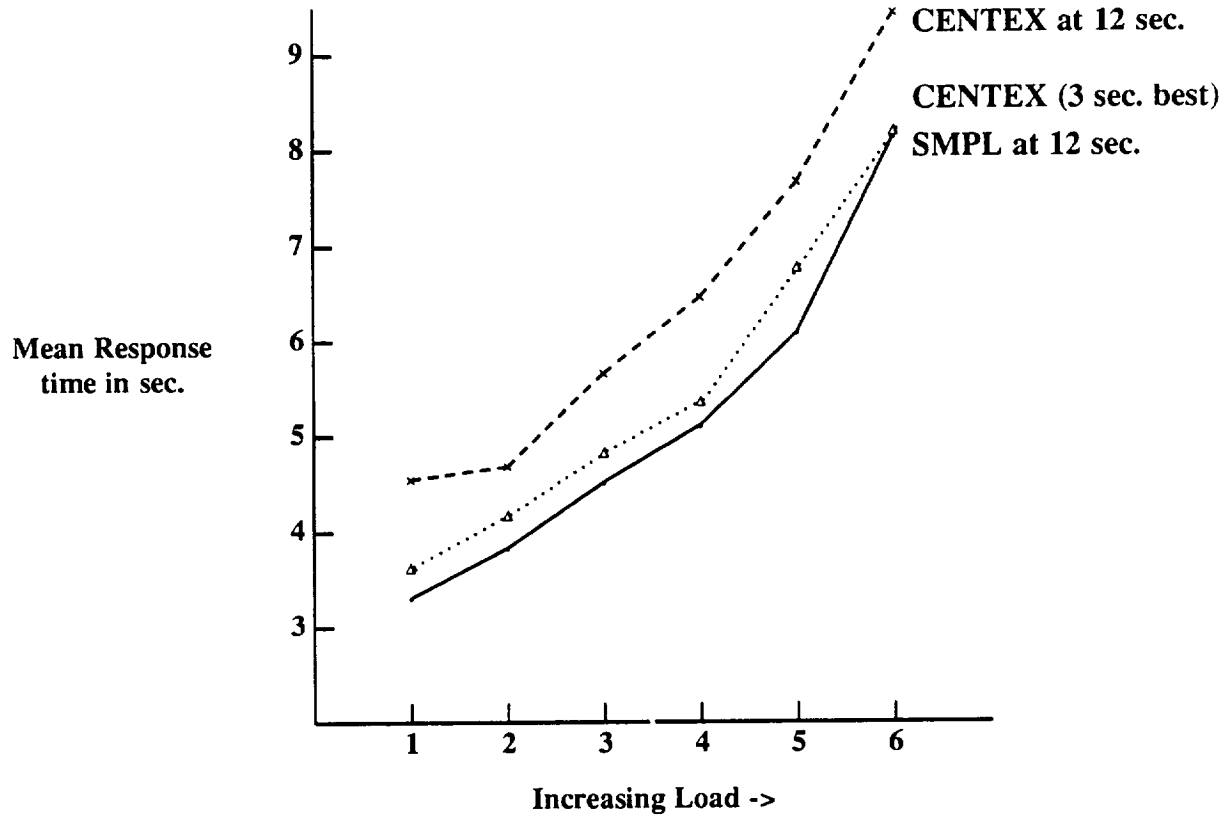

Figure 5b. SMPL using a four time slower update rate than CENTEX

at a processor. As seen in Figure 4, for very short intervals, the difference between the two heuristics becomes insignificant because the disadvantage of MINQ's coarse prediction is offset by the frequent update of status information.

Figure 5b compares the response times of SMPL (with a 12 second update interval) and CENTEX (with a 3 and a 12 second update interval)[4]. Even with an update interval which is four times as large, SMPL achieves response times that are up to 10% lower than that of CENTEX. As mentioned in the previous subsection, CENTEX does not sustain its performance when using large intervals because it lacks process specific information. SMPL used

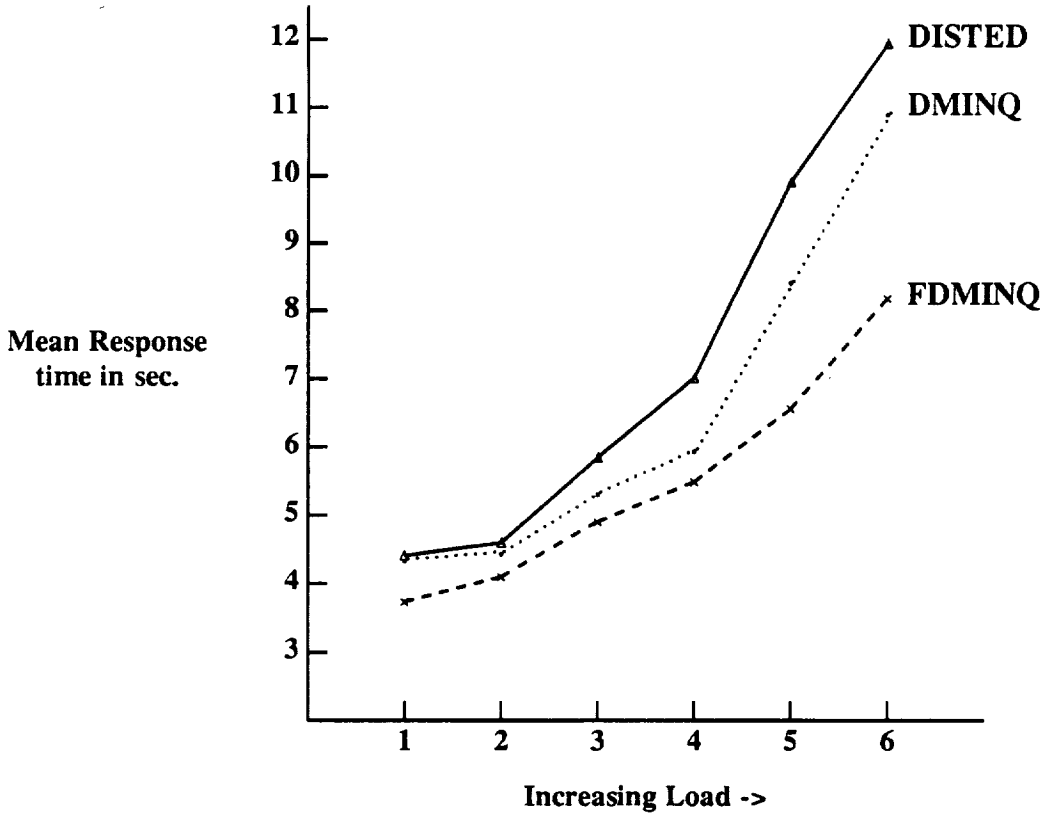

Figure 6. Comparison of DMINQ, DISTED (with T=2) & FDMINQ

[4]CENTEX achieves its best response time when using a 3 second update interval.

approximately 70% fewer status update messages than that used by CENTEX executed with a 3 second interval. Recall that only the CPU overhead of sending and receiving messages was modeled. Clearly, if the impact of message traffic was taken into account the results for SMPL would only improve more. This may be important since a recent study [Wallace 89] shows that an Ethernet channel in a distributed system can have sustained utilization of 50% or higher. In such an environment it is advantageous to have a heuristic that can perform well while imposing less message traffic on the system.

5.2 Comparison of the Distributed Heuristics

Figure 6 compares the response times of DISTED and DMINQ both of which use a threshold mechanism in making scheduling decisions. A threshold T equal to 2 was used in the experiment. This value was chosen because it produced the best response times for both heuristics. At low loads, both heuristics perform equally well since most of the processes are processed locally. However, as the load increases resulting in more processes being scheduled remotely, DMINQ out performs DISTED by up to 18%.

Figure 6 also contains the response time curve for FDMINQ. Recall that FDMINQ uses a filtering scheme to identify and execute small processes locally. Here, all processes with a *predicted* CPU time of less than 2 seconds were filtered. FDMINQ's response time was up to 51% (average 21%) lower than that of DISTED and up to 33% (average 17%) lower than that of DMINQ. Since the only difference between FDMINQ and DMINQ is the filtering mechanism, it is clear that explicit filtering of small processes is the reason for the improved response times.

Figure 7 shows the response time of *only* those processes that required less than half a second of CPU time. As a result of filtering, these processes execute up to three or four times faster at high loads than when DMINQ or DISTED is used. In addition, the variance in the

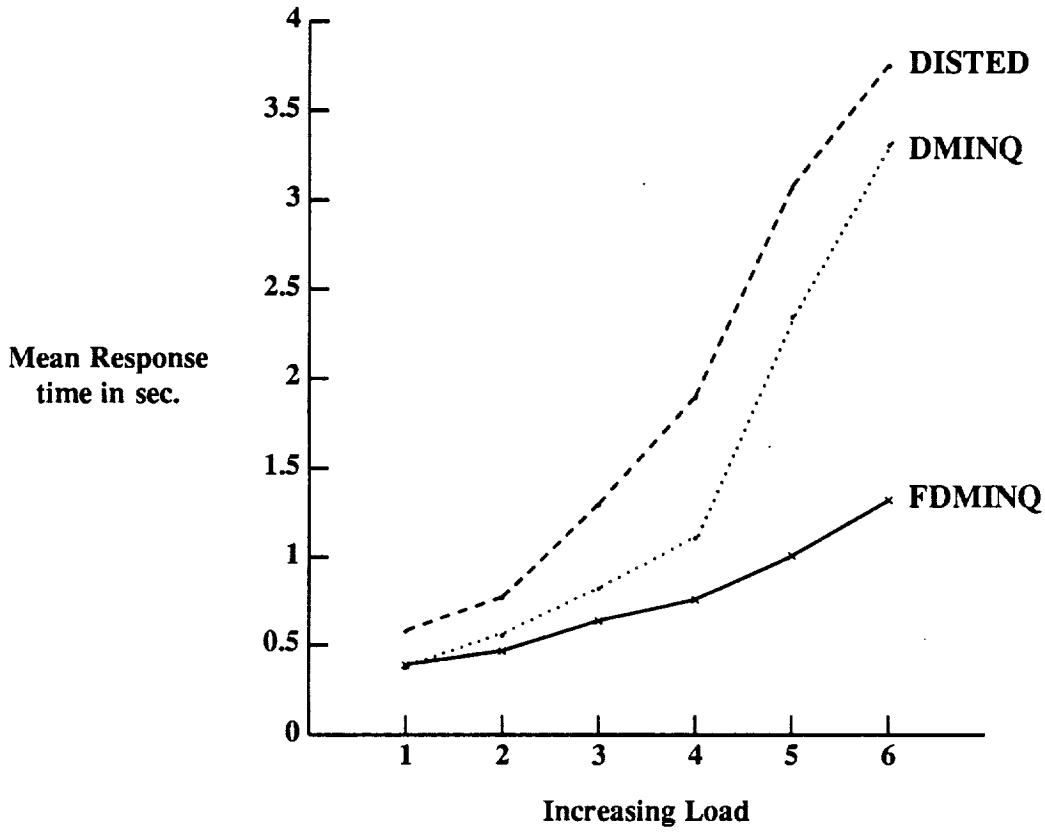

Figure 7. Response times of the small processes (< 0.5 CPU sec)

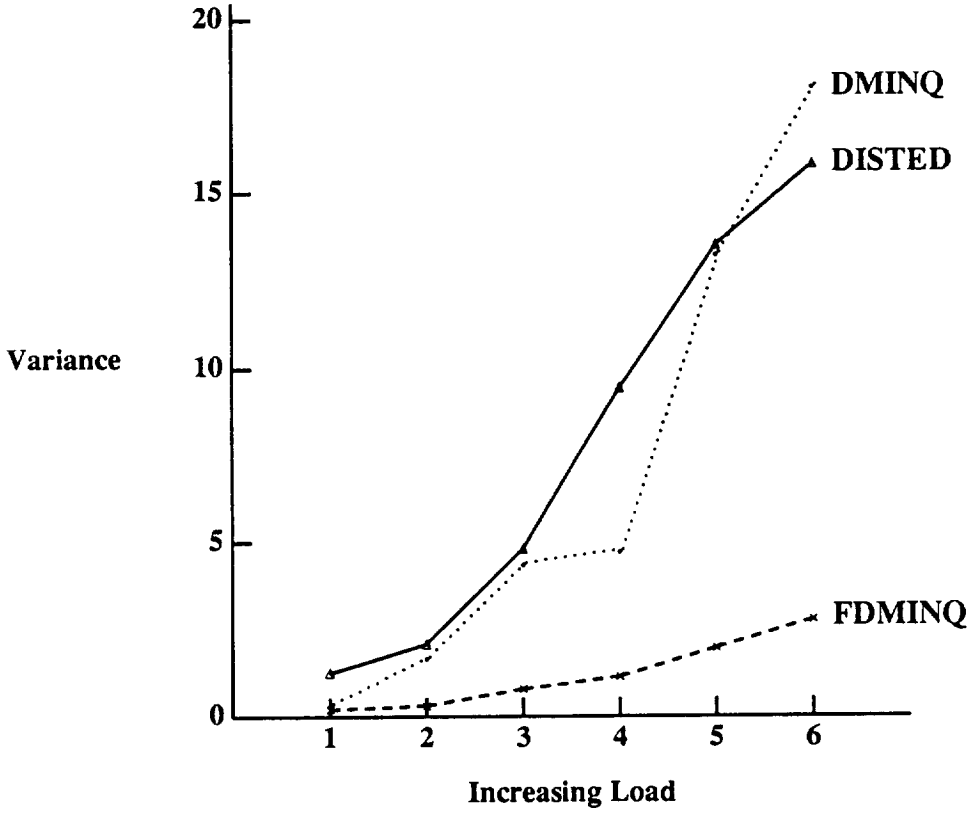

Figure 8. Variance in the response times of small process (< 0.5 CPU time)

response times of these processes is also reduced (Figure 8). It's especially important to maintain consistently low response times for small processes because even slight increases in their response times are noticeable to the user. The filtering scheme affects a large segment of the process population; 67% of the processes in the trace file require less than half a second of CPU time.

The threshold mechanism executes processes locally when a processor's load is below a threshold, T. As a result, it becomes ineffective if the processor's load is consistently higher than T. Figure 9 illustrates this behavior. However, the filtering mechanism always filters the same number of processes, regardless of the load. This is important because as the system load increases it becomes more efficient to execute processes locally. For this reason FDMINQ per-

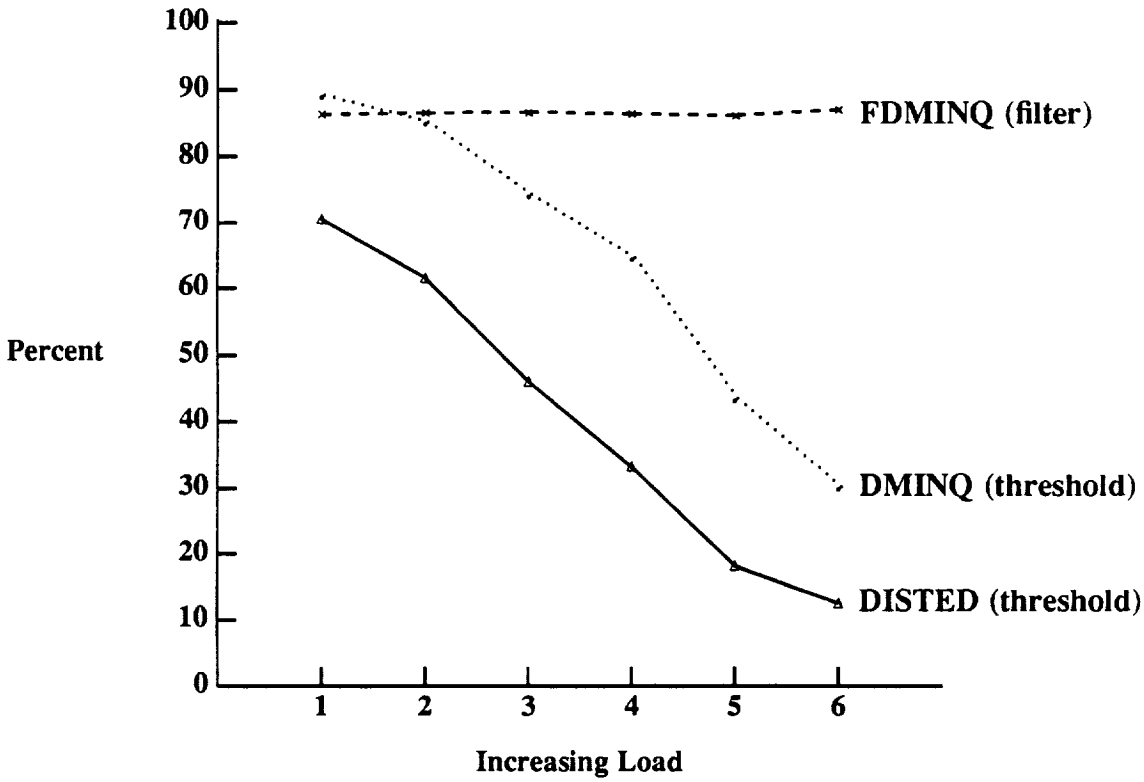

Figure 9. Percent of processes housed locally

forms better than DISTED and DMINQ with rising load (see Figures 6 and 7).

6.0 Conclusion

This paper has proposed new heuristics for load-sharing which use predicted information on process resource usage to make scheduling decisions. Four heuristics were presented. The first two, MINQ and SMPL, are centralized heuristics and the remaining two, DMINQ and FDMINQ, are distributed heuristics. These heuristics were first compared against random scheduling and then against two conventional heuristics, CENTEX and DISTED, which schedule processes solely based on system state information.

Results based on trace-driven simulations showed that the proposed centralized heuristics offer improved mean response times and are less dependent on the status update rate (the rate at which status information is collected). The simulations revealed that MINQ and SMPL perform as well or better than CENTEX while using up to 70% fewer status update messages. The use of fewer status update messages imposes less overhead on the system. In experiments where an equal number of status update messages were used, SMPL response times were up to 30% (average 22%) lower than those produced by CENTEX and, MINQ response times were on the average 18% lower. The use of prediction for distributed scheduling produced similar results. Under moderate to high loads, the response times for DMINQ were 18% lower than those of DISTED. When prediction was used to filter small processes and execute them locally a 50% improvement in response times was obtained.

Further study includes using prediction for a class of distributed heuristics that use probing such as the *Shortest* heuristic proposed in [Eager 86] and investigating the usefulness of prediction in real-time scheduling.

7.0 Acknowledgments

This work was supported by the National Aeronautics and Space Administration under NASA grant NAG-1-613. The authors would like to thank Murthy Devarakonda for his useful comments and suggestions. Particular thanks are due to In-hwan Lee and Bob Dimpsey for many helpful discussions.

8.0 References

[Barak 85]
A. Barak and A. Shiloh, "A Distributed Load Balancing Policy for a Multicomputer," *Software - Practice and Experience*, Vol. 15, September 1985.

[Casavant 88]
T. Casavant and J. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Trans. on Software Eng.*, Vol. 14, No. 2, February 1988.

[Devarakonda 89]
M. Devarakonda and R. K. Iyer, "Predictability of Process Resource Usage: A Measurement-Based Study of UNIX," *IEEE Tran. on Software Eng.*, Vol. 15, No. 12, December 1989.

[Eager 86]
D. Eager, E. Lazowska, and J. Zahorjan, "A Dynamic Load Sharing in Homogeneous Distributed Systems," *IEEE Trans. on Software Eng.*, SE-12, 5, May 1986, pp. 662-675.

[Goswami 89]
K. Goswami, R. Iyer, and M. Devarakonda, "Load Sharing Based on Task Resource Prediction," *Proceedings 22nd Annual Hawaii International Conference on System Sciences*, Volume 2, January 1989, pg. 921-927.

[Hwang 82]
K. Hwang, W. Croft, G. Goble, B. Wah, F. Briggs, W. Simmons and C. Coates, "A UNIX-Based Local Computer Network with Load Balancing," *IEEE Computer*, Vol. 15, 4, April 1982.

[Krueger 84]
P. Krueger and R. Finkel, "An Adaptive Load Balancing Algorithm For a Multicomputer," *University of Wisconsin, Technical Report #539*, April 1984.

[Lazowska 84]
E. Lazowska, J. Zahorjan, D. Cheriton, and W. Zwaenepoel, "File Access Performance of Diskless Workstations," *University of Washington, Technical Report 840-06-01*, June 1984.

[Leland 86]
W. Leland and T. Ott, "Load Balancing Heuristics and Process Behavior," *ACM*, September 1986.

[Livny 83]
M. Livny, "The Study of Load Balancing Algorithms for Decentralized Distributed Processing Systems," *Ph.D. Thesis, Weizmann Institute of Science*, August 1983.

[Ni 82]
M. L. Ni, "A Distributed Load Balancing Algorithm for Point-to-Point Local Computer Networks," *Proceedings of CompCon, Computer Networks*, Sept. 1983, pp. 116-123.

[Powell 83]
M. L. Powell and B. P. Miller, "Process Migration in DEMOS/MP", *Operating System Review*, Vol. 17, No. 5, 1983.

[Schwetman 86]
H. Schwetman, "CSIM: A C-Based, Process-Oriented Simulation Language," *Proceedings Winter Simulation Conference*, 1986

[Stankovic 85]
J. Stankovic, "Stability and Distributed Scheduling Algorithms," *IEEE Trans. on Software Eng.*, Vol. SE-11, No. 10, October 1985.

[Wallace 89]
R. B. Wallace and S. Zhou, "Network Performance in a Workstation Environment," *Proceedings 22nd Annual Hawaii International Conference on System Sciences*, Volume 2, January 1989, pp. 914-920.

[Wang 85]
Y. Wang and R. Morris, "Load Sharing in Distributed Systems," *IEEE Trans. on Computers*, Vol. C-34, No. 3, March 1985.

[Zhao 87]
W. Zhao, K. Ramamritham and J. A. Stankovic, "Scheduling tasks with Resource Requirements in Hard Real-Time Systems", *IEEE Trans. on Sofware Eng.*, Vol. SE-13, No. 5, May 1987.

[Zhou 86]
S. Zhou, "A Trace-Driven Simulation Study of Dynamic Load Balancing," *University of Cal. Berkeley, Technical Report #UCB/CSD 87/305*, September 1986.

[Zhou 86b]
S. Zhou, "An Experimental Assessment of Resource Queue Length as Load Indices," *University of Cal. Berkeley, Technical Report # UCB/CSD 86/298*, April 1986.