# Yale University
# Department of Computer Science


**Using Entropy Minimax Estimation for Load Sharing**


Ravi Mirchandaney, Lui Sha and John A. Stankovic



YALEU/DCS/TR-601
January 1988

This work has been supported in part by NSF under grant ECS-8406402, by RADC under contract RI-44896, and by ONR under Grant N00014-86-K-0564

# Using Entropy Minimax Estimation for Load Sharing*

Ravi Mirchandaney†
Dept. of Computer Science
Yale University
New Haven, CT 06520-2158

Lui Sha
Computer Science Dept.
Carnegie-Mellon University
Pittsburgh, PA 15213

John A. Stankovic
Dept. of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

January 1988

**Abstract**

In this paper, we study the problem of load sharing in distributed systems when job transfers and remote state updates encounter significant delays. In particular, we are interested in on-line parameter estimation for threshold based load sharing algorithms under these conditions. The specific parameter of concern is the operating threshold. The load sharing algorithms actively acquire remote state information. Because it is assumed that significant delays are encountered during job transfers and state updates, there is uncertainty in the remote observations. The estimation technique that is used to reduce this uncertainty and determine the appropriate threshold is called Entropy Minimax. It is seen that Entropy Minimax possesses several useful properties relevant to this problem. We study the effect of varying the threshold on four load sharing algorithms under different conditions of loads, delays and service distributions. It is shown that in most of the cases, the threshold selected by Entropy Minimax produces the optimal performance. We also study the effects of job transfer and probing overheads on the four algorithms.

*This work was supported, in part, by the National Science Foundation under grant ECS-8406402 and by RADC under contract RI-44896X.

†Part of this work was performed while the author was at Carnegie-Mellon University.

# 1 Introduction

There have been many studies regarding load sharing (balancing) in distributed systems. Many of these efforts have concluded that simple load sharing algorithms produce significant performance improvements over no load sharing. Researchers have utilized several different techniques to study load sharing. Theoretical studies have predominantly employed queueing methods, with some studies being based upon network flow algorithms and optimization techniques [7], [6], [17], [16]. There have been several studies performed using simulation analysis and the proliferation of experimental distributed systems have given rise to implementations of load sharing algorithms on real machines [10], [15], [1], [18]. Some other relevant references include [3], [12], [9], [2], [14]. For a more detailed summary of load sharing research, the reader is referred to [11].

Some of these above mentioned efforts discuss the performance of simple threshold based policies [7], [12]. These papers provide significant insight into the relation between the performance metrics and the various important parameters of load sharing algorithms. It is seen that the optimal threshold is a function of the system load, among other factors. However, these papers do not discuss the issue of determining the threshold on line, which is particularly relevant in cases where the load at a node is not explicitly made available to the job scheduling algorithm or the load changes over time. It has been shown in earlier studies [12], [11] that using a sub-optimal threshold may degrade the performance by 10-20% in some cases and hence we feel it is interesting to address this problem. In this paper, we study a method to estimate the threshold as a function of average job transfer delay (known explicitly beforehand) and load (which is unknown).

The main contributions of this paper include the application of Entropy Minimax, an information theoretic estimation technique that reduces the uncertainty of remote state information arising due to the delays. We study the performance of the algorithms over a large range of loads and delays. The effect of varying the threshold is determined for three service distributions. We also examine the effects of including job and probe overheads on the load sharing algorithms. Some of the interesting results that we have obtained are as follows: It is shown that the thresholds generated by Entropy Minimax produce optimal or near-optimal performance. This is valid for three different service distributions over a large range of parameter values. In cases where the Entropy Minimax threshold does not produce optimal results, the performance difference between using the optimal threshold or the one generated by Entropy Minimax is very small. It is shown that when the loads are high, the benefits from load sharing are more significant for a larger range of delays. The performance improvements due to load sharing are higher when the variability in the service distributions increases. As regards probe and job overheads are concerned, high costs for these parameters adversely affect some algorithms more than they do others (details in Chapter 3). Further, the performance relations between the algorithms change as the costs are varied.

The remainder of this paper is organized as follows: In Section 2, we discuss the motivation for this research. The relation between delays and state classification is also addressed in Section 2, with a brief description of Entropy Minimax. In Section 3, we address the question of load sharing in systems where the algorithms utilize thresholds that are generated by Entropy Minimax. It is observed that these thresholds are able to provide optimal response times for most of the cases tested. In Section 3, we study load sharing under exponential as well as non-exponential service time distributions. We also determine the effect of probe and job transfer overhead costs on the performance of the load sharing algorithms. Finally, in Section 4, we summarize the main contributions of this paper.

# 2 Delays and Load Sharing

## 2.1 Motivation for Entropy Minimax

We assume that the system consists of $N$ identical nodes where each node possesses a network controller that is responsible for most of the overhead associated with load sharing activities. Thus, although the CPU overhead due to load sharing is assumed to be negligible, job transfers and state update messages are delayed because of processing by the network controller. Jobs are executed on a First-Come-First-Served basis. Also, the minimization of the number of idle nodes while some other nodes have waiting jobs is the main objective of the load sharing algorithms discussed in this paper. This objective follows from the discussion by Livny and Melman [10] where it was shown that significant performance degradation occurs when idle nodes coexist with nodes that possess waiting jobs.

The optimal strategy when the transferred jobs encounter negligible delays is to exactly emulate the M/M/n behavior. However, this may not be the right strategy to utilize in the face of delays. For example, suppose a node becomes idle. It tries to acquire a remote job by probing the other nodes in the system. Because of the job transfer delays, the node remains idle until the remote job arrives (unless it receives a local job). Obviously, performance can suffer for this period. If the delays are non-trivial (for example, on the order of one service time or more), the degradation in performance can be severe.

It may then become necessary to try and acquire a remote job even before a node becomes idle. The question then is: When should the process of searching for a remote job begin? Is it possible to determine the correct subset of states which corresponds to the idle state in the negligible delay case? An estimation technique that appears to possess the suitable characteristics in this regard is called Entropy Minimax [5], [4], which has its roots in Shannon's theory of communication.

One of the main problems associated with distributed algorithms which actively ac-

quire remote state information is related to the uncertainty in this remote information. Entropy Minimax has the intuitive appeal of extracting *all but no more* than the truly available information from the remote and delayed observations [4][1]. Furthermore, this method has the advantage of being non-parametric. This is particularly relevant in our context, because the very act of load sharing transforms an initial known arrival distribution into some unknown distribution, thus limiting the applicability of parametric estimation methods.

Broadly speaking, Entropy Minimax utilizes two meta-level states, as follows: One is called Low, which indicates that a remote job is needed in order to prevent the node from becoming idle, and the other is called High, which indicates that a job is not needed. The intuition behind using this method is to find a threshold which partitions the state description (in this case the queue length) at each node into these two meta-level states, High and Low, as a function of delay and load. This partition has the property that when a job is transferred from a High node to a Low node, the probability that it arrives when the destination node just becomes idle, is maximized.

## 2.2 Entropy Minimax

Conceptually, Information

$$I(x) = \log \frac{1}{p(x)}$$

is a measure of the uncertainty associated with the outcome of a random variable $X = x$. If the outcome is certain i.e., $p(x) = 1$, then we have no information gain from observing the outcome and $I(x) = 0$. On the other hand, if the outcome is very uncertain, i.e., $p(x) << 1$, then the information gain from observing the outcome is large. Entropy $H(X)$ is the average information associated with a random variable $X$, i.e.,

$$H(X) = \sum_{i=1}^{n} x_i \log \frac{1}{p(x_i)}$$

The concepts of conditional information, $I(X|Y)$ and conditional entropy $H(X|Y)$ follow in a similar manner. If there are no errors associated with the observation of the events [8], the outcome of the random variable $Y$ perfectly determines the outcome of the random variable $X$, then $P(X = x|Y = y) = 1$ and we have $H(X|Y) = 0$. Since $H(X|Y) = 0$ is in general unattainable, Entropy Minimax attempts to minimize its value.

Let us now try to see how this is related to the problem of estimation in the context of load sharing. If $\tau$ is the delay in acquiring a job, can we determine a threshold $T$ such that the following criteria hold?

[1]For a detailed comparison between this and other well known estimation techniques like maximum likelihood estimation, least squares etc, see [4]

If $N_i(t) \leq T$, then $N_i(t+\tau) = 0$, where $N_i(t)$ is the number of jobs at node $i$ at time $t$, and,

If $N_i(t) > T$, then $N_i(t+\tau) > 0$.

The above requirements are for an idealized threshold such that the node is able to perfectly predict its future state, in case no action is taken by it to try and acquire a remote job. The threshold serves the purpose of classifying the states into two meta-level states, one corresponding to Low (which includes all states to the left of $T$ as well as $T$, i.e., $0 \ldots T$) and the other corresponding to High (which includes all states to the right of $T$). While it would be ideal to possess such a threshold that acts as a perfect predictor for future states, it may not be achievable in practice, because most systems that we study are stochastic in nature. Thus, the question is whether this idealized threshold can be approximated to any degree such that in a majority of the cases, the prediction is correct. In other words, on the average, a requested remote job arrives when the requesting node becomes idle.

We now describe the Entropy Minimax algorithm that is used to compute the threshold. Let the nodes have local arrivals which are serviced in a FCFS manner. There is no load sharing being performed at this stage. Every $\tau$ units of time (average job transfer delay), the number of jobs in the node is recorded. Associated with each state $l$ (queue length), are two counters, $a_l$ and $b_l$. If the node is idle at time $t$, the $b_l$ counter corresponding to the number of jobs $l$ in the node at time $t - \tau$ is incremented by one. In any case, the $a_l$ counter corresponding to the number in the node at time $t - \tau$ is incremented by one. Basically, what this process does is provide a set of conditional probabilities as follows. We define

$$p_i = P(N_{t+\tau} = 0 | N_t = i),\ 0 \leq i \leq T_m$$

as the probability that the node becomes idle at time $t + \tau$, given that its queue length at time $t$ is equal to $i$ and $T_m$ is the the maximum number of states needed. The conditional probabilities are computed as follows:

$$p_i = b_i / a_i,\ 0 \leq i \leq T_m$$

where $i$ is a particular value of the queue length. These conditional probabilities are collected for the states (queue lengths) that a node enters. Let the instantaneous threshold (during the Entropy Minimax computation stage) be $l = [0, T_m]$. In practice, it is observed that no more than 8-10 states are really needed, for the loads and delays that we have studied. Thus, only $T_m + 1$ partitions of the queue lengths are examined by the Entropy Minimax algorithm [4].

For a threshold $l$, the following sums are defined:

$$B_l(L) = \sum_{i=0}^{l} b_i$$

$$A_l(L) = \sum_{i=0}^{l} a_i$$

$$B_l(R) = \sum_{i=l+1}^{T_m} b_i$$

$$A_l(R) = \sum_{i=l+1}^{T_m} a_i$$

where $0 \leq l \leq T_m$ and $L$ and $R$ denote the left and right partitions respectively, generated by the threshold $l$. The conditional probabilities associated with the partitions are:

$$P_l(L) = B_l(L)/A_l(L)$$

and

$$P_l(R) = B_l(R)/A_l(R).$$

The logarithm of each of these conditional probabilities is the conditional information received from the observation. The expected value of the information is the conditional entropy of observation, $H(X|Y = l)$, when the instantaneous threshold $= l$.

$$\begin{aligned} S_l(L) &= -P_l(L)\ln P_l(L) - (1 - P_l(L))\ln(1 - P_l(L)) \\ S_l(R) &= -P_l(R)\ln P_l(R) - (1 - P_l(R))\ln(1 - P_l(R)) \\ H(X|Y = l) &= P(L)S_l(L) + (1 - P(L))S_l(R) \end{aligned}$$

where $P(L) = \frac{A(L)}{A(L)+A(R)}$. The threshold is computed using the following optimization procedure:

$$\text{minimize} \quad H(X|Y = l) \tag{1}$$

subject to simple probabilistic constraints as described in [4].

The Entropy Minimax algorithm steps through the queue lengths, starting at zero. The conditional entropy is computed at each step and the queue length corresponding to the minimum entropy is selected as the threshold. What this threshold implies is that if a node reaches this threshold, it is very likely to become idle at time $t + \tau$ unless a remote job is made available to it. Further, if the queue length is greater than the threshold, the likelihood is that the node will *not* become idle in the given interval of time. In order to determine the relation between delays, load and state classification, we have conducted several experiments. The results and their implications are described in the following subsection.

Table 1: Results from Threshold Experiments

| Delay | 0.0 | 0.5 | 1.0 | 1.5 | 2.0 | |
|---|---|---|---|---|---|---|
| Load | | | | | | Dist |
| | 0 | 0 | 0 | 1 | 1 | Exp |
| 0.4 | 0 | 0 | 1 | 1 | 1 | Erl |
| | 0 | 0 | 0 | 1 | 1 | Hyp |
| | 0 | 0 | 1 | 1 | 1 | Exp |
| 0.6 | 0 | 0 | 1 | 1 | 1 | Erl |
| | 0 | 0 | 1 | 1 | 2 | Hyp |
| | 0 | 1 | 1 | 2 | 2 | Exp |
| 0.8 | 0 | 1 | 1 | 1 | 2 | Erl |
| | 0 | 1 | 1 | 2 | 2 | Hyp |

## 2.3 Results from Threshold Experiments

In this section, we present the results from simulation studies which specifically address the issue of the threshold and the two factors that affect it greatly, namely delays and the load. The algorithm that computes the threshold samples the queue length of the node at intervals of time $= \tau$ (the average job transfer delay). The conditional probabilities are computed from the above observations, as described in the previous subsection. The threshold is computed after a large enough number of samples have been gathered. This can be verified by noting whether the threshold has stabilized. It was seen that if $S$ was the expected service time for jobs, then a window of about $400S$ to compute the threshold appeared to be adequate. Table 1 depicts the variation of the threshold as computed by Entropy Minimax, over a range of delays and loads, and for three different service time distributions (with identical means), i.e., exponential (Exp), 2 stage Erlang(Erl) and hyperexponential(Hyp) with $C_v^2 = 2.0$. The utilizations tested were 0.4, 0.6 and 0.8 and the average job transfer delays were $0.5S$, $S$, $1.5S$ and $2S$, for each of the loads tested.

From Table 1, it can be seen that:

- The threshold is a non-decreasing function of delays. This is because greater delays increase the uncertainty in the information, leading to a greater aggregation of low states, represented by the higher thresholds.
- As the delays tend to zero, the threshold approaches zero for all loads. This is intuitively satisfying because at insignificant delays, it is appropriate to emulate the optimal M/M/n strategy. A threshold of zero makes this possible.
- For a given delay, higher loads tend to increase the threshold, because the holding time of a state decreases with the increase in load.

- In some cases, the thresholds appear to be dependent upon the service time distributions, as may be seen in Table 1 (e.g., $\rho = 0.6$, delay $= 2S$). However, the exact relation between distributions and thresholds is not clear.

In this paper, we restrict our study to the situation under which delays are significant but remote state information is still useful in the sense that a stable Entropy Minimax threshold can be found. In extremely high delays, we have seen that the threshold is undefined[2]. Furthermore, at such high delays, it may be the case that load sharing will actually make the performance worse because the transferred jobs will be in neither the sender's queue nor the receiver's queue for the duration of the job transfer.

# 3 Simulation Studies

## 3.1 Description of Algorithms

The algorithms that we have utilized in this paper are described in the following few paragraphs. Each node is provided with a threshold $T$ and probe limit $L_p$.

**Symmetric:** As soon as a node's queue length goes below $T+1$ on the completion of a job and the node is not already waiting for a remote job, it probes $L_p$ nodes in the system, until a node can provide it with a job or all the nodes have been exhausted. If more than one node can transfer a job, one of these nodes is selected at random. A remote node will only transfer a job if it possesses at least $T+2$ jobs. Also, as soon as a local arrival occurs at a node and it has at least $T+2$ jobs (including the new arrival), it probes $L_p$ nodes in the system, until it finds a node which has $\leq T$ jobs and is not already waiting for another remote job. If all the probed nodes have at least $T$ jobs, no transfer will take place. If more than one node can accept a spare job, only one of these will be selected at random for transfer.

**Forward:** If a local arrival occurs and the node has at least $T+2$ jobs (including the newly arrived one) it probes $L_p$ nodes to determine if any one is $\leq T$ and is not already waiting for a remote job. If so, it transfers this job there, else, it keeps this job. If more than one node is able to accept a spare job, one of these nodes is selected at random for transfer.

**Reverse:** As soon as a node goes below $T+1$ on the completion of a job and it is not already waiting for a remote job to arrive, it probes $L_p$ nodes to determine if any node has a spare job (at least $T+1$), the remote node transfers a job to this node. If more than one node responds positively, one of these nodes is selected at random.

[2]In these cases, the conditional entropy associated with different threshold values are approximately equal to each other.

**Random:** In this algorithm, the nodes do not perform any probing. As soon as a node receives a local job, it checks if it has at least $T + 1$ other jobs. If so, it transfers this new job to one of the other nodes, selected at random. There is no state update overhead generated by this algorithm and it serves to provide a reasonable bound for comparison against probing algorithms.

## 3.2 Description of the Experiments

The simulation system, which was written in SIMSCRIPT II.5, consisted of 10 identical nodes, connected in a network. The inter-arrival and job transfer times were exponentially distributed. However, the service times were selected from three different distributions, depending upon the test being conducted. These were, exponential, 2-stage Erlangian and hyperexponential distributions. Further, the arrival and service rates were identical at the nodes. The thresholds for the various loads, delays and service distributions (see again Table 1) were computed off-line by the Entropy Minimax program. This was convenient at the time, but the computations could as well have been performed on-line. The delays were varied from 0.5*S to 2*S, where $S$ the expected service time of jobs, was 1.0 units. The load was varied from 0.4 to 0.8, which encompassed a large range for load sharing.

Every time a job was transferred, it immediately disappeared from the sender's queue and appeared at the receiver's queue after the transfer delay, which was exponentially distributed. Thus, a transferred job was not available for execution at either node during this interval. While jobs encountered delays, probes were assumed to take zero time. It has been shown elsewhere [13] that as long as jobs take at least ten times the amount of time to reach their destination as compared to probes, this assumption is quite reasonable. Although it was possible to exactly determine the state of remote node at the time of decisions, the delay in actual job transfer caused the uncertainty in the state information, because the load sharing decisions were based upon states that may have changed by the time a transferred job arrived at the remote node.

The maximum number of probes that a node was allowed to make was tuned initially as a parameter. It was seen that 2 was a good number in most instances, in a 10 node system, particularly if the probe overhead would be accounted for in some way. The incremental gain in performance by allowing $L_p = 3$ over 2 was marginal in all cases tested. Experiments conducted with complete probing ($L_p = 9$) showed very little improvement over $L_p = 2$ or 3 [11]. Thus, the simulation runs in the following section were made with 2 probes. Also, the nodes to be probed were selected at random.

## 3.3 Results

In this section, we present the results that we have obtained from simulations conducted on a network of 10 nodes. The main metric of interest is the average response time generated by the algorithms. As was discussed in [12], the overhead generated by load sharing was assumed to be entirely transferred to the network controllers, except when we specifically address the issue of overhead costs. The results include the effects of thresholds on performance, the performance of the various algorithms under different loads and delays, the effects of various overhead costs for probes and job transfers and the study of load sharing with non-exponential service times.

All the results indicated are averages of at least three independent simulation runs with different random number seeds (i.e., the method of independent replications). It was seen that in typical cases, the sample standard deviation was less than 0.2%, and the 95% confidence interval lay between $\pm 0.3\%$ of the sample mean. These confidence intervals were computed using the Student-t distribution. Unless otherwise stated, the service times are assumed to be exponentially distributed.

### 3.3.1 Choosing an Algorithm

Figures 1, 2 and 3 depict the performance of the four algorithms under delays for the loads of 0.4, 0.6 and 0.8 respectively. In general, one can see that Forward with 2 probes performs well over the range of parameters tested. For the case of low loads, Random is very effective and generates only slightly worse response times than Forward. However, there is point of discontinuity at $\rho = 0.8$ and delay = 0.0. It is seen that the performance of Random improves when the delay in increased to 0.5*S. This counter-intuitive behavior has to do with the fact that in general, Random does not fulfill the requirement imposed by the Entropy Minimax state representation that a transfer should take place from a High node to a Low node. This is particularly true at $\rho = 0.8$ and zero delay where a transferred job is likely to arrive at a High node quite often. However, at higher delays, the threshold increases and so does the probability that a transferred job will arrive at a Low node. In other words, the threshold that might be optimal for probing algorithms may not necessarily be optimal in the case of Random assignment, particularly at low to moderate delays.

Reverse does not perform as well as Forward, particularly when the load is $\leq 0.6$. This is because more nodes are likely to be in the Low state, making it harder for Reverse to find a spare job. However, in zero delay experiments with extreme ($\rho \geq 0.9$) loads, Reverse performed better than Forward (observed in [12]).

It can be seen from Figures 1, 2 and 3 that as the job transfer delays increase, the following observations may be made: Firstly, the benefits of load sharing become less significant and the performance under load sharing approaches that of the $M/M/1$ system. In fact, the response time may actually get worse than $M/M/1$ at even higher

transfer delays and we have seen this occur in other experiments (results not presented here). This is particularly evident at $\rho = 0.4$ (Figure 1) where at delay $= 2S$, the performance under load sharing is only about 5% better than the $M/M/1$ response time at that load. On the other hand, at $\rho = 0.8$, the benefits of load sharing are more substantial even at the maximum delay tested (Figure 3). It is seen that for delay $= 2S$, the performance under load sharing is about 45% better than the corresponding $M/M/1$. Thus, it is possible to hypothesize that at even higher loads greater delays may be tolerated. At low loads, it might make sense to turn off load sharing when the job transfer delays increase beyond $2S$. In comparison with the perfect load sharing with no overhead or delays ($M/M/10$) performance, the algorithms do not perform well, as might be expected. Furthermore, the performance of the probing algorithms become almost identical and at low loads, Random performs as well as any probing algorithm tested (at $\rho = 0.8$, there is about 5% improvement on account of probing).

### 3.3.2 Effect of Thresholds on Three Service Distributions

In [12], [13], we concentrated on the performance of the load sharing algorithms when service times are exponentially distributed. In general, the exponential assumption may be considered restrictive and we would like to see how appropriate are the Entropy Minimax thresholds when the service times are not exponentially distributed. In addition, we are interested in comparing the performance of load sharing for the three service distributions, i.e., exponential, 2 stage Erlangian and hyperexponential with $C_v^2 = 2.0$, all having the same mean, 1.0 units.

Figures 4, 5, 6 and 7 depict the behavior of the Symmetric probing algorithm for the three service time distributions discussed above. The arrival rates were 0.4 and 0.8 jobs/unit and the job transfer delays were $0.5S$, $S$, $1.5S$ and $2.0S$ in Figures 4, 5, 6 and 7, respectively. Results for $\rho = 0.6$ are not presented owing to space limitations in the graphs. The curves in the graphs represent not the actual response times obtained but the response times normalized by the corresponding no load balancing response times generated by the $M/M/1$, $M/H_2/1$ and $M/E_2/1$ values for the appropriate service time distributions. Thus, the results of load sharing with exponential service time are normalized by the $M/M/1$ values, the Erlangian by $M/E_2/1$ and hyperexponential by $M/H_2/1$. Further, the thresholds were varied between 0 and 5, and the response times under these conditions were recorded.

From this set of figures, we are able to make the following interesting observations:

- The improvement by load sharing over the corresponding no load sharing is greater as the load increases. The curves for $\rho = 0.4$ are located higher than those for $\rho = 0.8$ ($\rho = 0.6$ curves are located in between, although we have not depicted these in the graphs). For example, from Figure 5, delay $= S$, the best improvement for Hyp is 60% over no load balancing when $\rho = 0.8$ but only 25% when $\rho = 0.4$.

Similar numbers may be determined from the other three figures in this set. This occurs because of the higher inherent waiting times at high loads and the load sharing is consequently most advantageous.

- For most of the range of thresholds and delays tested, it seems that the improvements are greater when the variability in service rates is higher. Thus, in general, the curves for hyperexponential are located lowest, Erlangian are highest with exponential in the middle of these two (an exception is seen in Figure 4, $\rho = 0.8$ when the Exp curve crosses the Erl). For instance, in Figure 4, $\rho = 0.8$, the gain by Hyp is greater than 65% whereas Erl is better by about 50% over its no load balancing value. Again, this is because higher variability in service times implies longer waiting times. We postulate that similar behavior may be expected when different arrival distributions are examined.

- Varying the threshold over a large range of values helps us determine whether the thresholds predicted by Entropy Minimax are correct (i.e., they provide the optimal response times). In this connection, we refer back to Table 1 which encapsulates the Entropy Minimax thresholds for the loads 0.4, 0.6 and 0.8 for the 4 values of job transfer delays. From Table 1 and the graphs under consideration, it is seen that for most of the parameters tested, the thresholds generated by Entropy Minimax are optimal. In fact, out of the 36 different threshold tests conducted (results of which are depicted in Table 1), the optimal threshold as obtained by simulations of the Symmetric algorithm was different in only 3 instances. For example, for Erlang-2 service times with $\rho = 0.4$ and delay $= 2S$, the optimal threshold was 2 but Entropy Minimax predicted 1. However, the differential in response times using one or the other threshold is less than 2% (see 7, topmost curve).

### 3.3.3 Effects of Probe and Job Transfer Overheads

Thus far in this paper, the response times generated by the algorithms did not include the effect of probe and job transfer overhead. The reasons for doing this were as follows: Firstly, we had assumed that the overhead of processing jobs and probes for load sharing is completely transferred to the network controllers (BIU). In a perfect world, the CPU at a node would not be slowed down owing to interference from the network controller. Secondly, it was our conviction that it is very hard to estimate reasonable costs for potential interference since these costs are highly dependent upon the underlying system architecture and protocols and at this stage we have made only very general assumptions regarding the node architecture.

The response to the above arguments may be the following: In reality, network controllers will tend slow down the CPU to some extent because of the common resources they may need to access, e.g., shared memory, system bus and so on. System designers will consequently be interested in determining the effects that the interference may

have on the system performance as a whole. In order to provide a feel for how a range of overhead costs might affect the net response time of jobs, we have conducted several experiments where the effect of probe and job transfer overhead is modelled as an interference to the jobs executing on the CPU. We have chosen to use a simple interference model because it is our belief that these simplifications will affect all the algorithms equally and that the relative comparisons will consequently be unaffected. In short, to test the effect of probe overhead, we increased the CPU time requirement of the currently executing job after a certain number of probes were made by that node. A similar strategy was adopted to account for job transfer overhead, except that we kept track of the number of jobs transferred.

**Probe Overheads**

To study the effect of probe cost, each node in the simulation system accumulates the number of probes it sends out and receives. As soon as the number increases above a prescribed value, the next job to be executed at this node executes at a slower rate. The normal mean service time of jobs was 1.0 units, as in all the earlier experiments. However, the interference is assumed to increase the mean service time of the slower job to 1.25 units, an increase of 25%. The amount of increase is arbitrary and is used solely for the purpose of illustration. The counters which hold the probe counts are then reset and the monitoring process continues. When the effect of job transfer overhead is to be studied, the nodes accumulate the number of jobs transferred. When this count increases above a prescribed value, the next job executes at the slower rate and the job transferred count is reset to zero and the monitoring process continues.

Figure 8 depicts the results for the case of probe overheads (job transfers do not cause interference in these experiments). The horizontal axis represents the probe count at which the job execution time is increased. For instance, a count of 10 means that as soon as a node processes 10 probes, it schedules the next job to execute at the slower processing rate. This count is varied between 10 and 100, with 10 generating the highest overheads. $Sy, Re, Fo$ represent the results for Symmetric, Reverse and Forward probing respectively.

From Figure 8, we can make the following observations: At $\rho = 0.4$, the effect of probe overhead is most felt by Reverse probing. Most of the reverse probes do not result in job transfers and only contribute to overhead. As the probe costs decline, Forward and Symmetric show identical behavior. However, when probes costs are high, Symmetric performs slightly worse than Forward. Similar behavior is seen at $\rho = 0.6$ where Forward is better than Symmetric at high probe costs, because Symmetric generates a large number of wasted reverse probes. However, at low loads, Reverse is inherently worse than Forward or Symmetric in the first place. At $\rho = 0.8$, Reverse does better than Forward and Symmetric at high probe costs, by about 20% when the probe costs are the highest. This is because nodes are more likely to make forward probes at this load and high costs of probes will adversely affect Forward and Symmetric probing. When probe costs are low, Symmetric is clearly superior to either Forward or Reverse and even

Forward ends up being slightly better (about 2%) than Reverse at the lowest probe costs tested.

**Job Transfer Overheads**

In the next set of tests, each node monitors the number of jobs it sends out and receives. As soon as the number increases above a prescribed value, the next job to be executed at this node executes with the slower rate. The counters which hold the job counts are then reset and the monitoring process continues. In Figure 9, we present the results of the experiments wherein probes generated no overhead. Instead, job transfers causes interference to the currently executing CPU job and increases the average service time of the delayed job by 25%, as in the case of the previous experiments on probe overheads. The horizontal axis in Figure 9 represents the job count at which the execution time is increased. This count is varied between 1 and 25, with 1 generating the highest overheads. The significance of the job count is similar to the probe count in Figure 8. As soon as the job transferred count is reached, the next CPU job is executed at the slower speed. For comparison purposes, we have included the curves for the Random assignment algorithm in the presence of job transfer overheads. These curves are represented by the code *Ra*. The other codes are the same as those for Figure 8..

From Figure 9, we can make the following observations: The effects of job overheads is most felt by Symmetric and Random. This is because they tend to transfer the most number of jobs, especially as the loads increase. This is shown by the fact that extremely high (with the potential of becoming unstable) response times are generated by Random at $\rho = 0.8$ and high job overheads. While Symmetric is stable, it does perform worse than the other two probing algorithms at $\rho = 0.8$, when the overheads are relatively high (about 5-10% worse). As the overhead costs decrease, the behavior becomes more predictable with Symmetric clearly performing better than all the other algorithms.

To reiterate our earlier reservations about assigning overhead costs, we believe that these costs are very tightly linked to the underlying system architecture. Designers of such systems have to estimate the potential interference that might be caused by probes and job transfers, given the underlying node architecture. If estimates of these quantities are available, then Figures 8 and 9 can provide some help in selecting the algorithm that might be most appropriate. For instance, if probe costs are very high and the load is $\geq 0.8$, it might be appropriate to select Reverse probing. At low loads ($\leq 0.7$) and high probe costs, Forward appears to be the best bet. If however, the probe costs are low, Symmetric will easily outperform either of the other two probing strategies. For high job costs and high loads ($\geq 0.8$), Symmetric is definitely worse than either of the probing algorithms. However, when job costs are not very high, it is probably best to go with Symmetric. This fact was also noted in [12], where Symmetric outperformed Forward and Reverse when overhead costs were zero.

# 4 Summary and Conclusions

This study was primarily concerned with studying the effects of delays in load sharing. We presented an application of Entropy Minimax, an information theoretic estimation technique, to reduce the uncertainty in the delayed state information. It was seen that the performance of the algorithms using the state classification provided by Entropy Minimax was for the most part optimal. In the few instances that the thresholds were not optimal, the response times differed from the optimal results by at most a few percent. This fact was verified for a large range of parameter values as well as three different service distributions: exponential, 2 stage Erlang and hyperexponential. However, it was seen that the performance of the algorithms is less sensitive to appropriate selection of operating thresholds than we had originally imagined. If the chosen threshold differed by one from the optimal threshold, the performance could be worse by 5-10%.

As regards the state update protocol and algorithm design, it was seen that Symmetric probing with probe limit of 2 probes performed uniformly well over the range of parameters tested with Forward being next best. For higher values of load ($\geq 0.9$), Reverse is likely to outperform Forward, as we have seen in the results of [12]. We studied the effects of probe and job transfer overheads on the CPU service rates and concluded that for high overhead costs, Symmetric performs poorly because it has the tendency to generate large numbers of probes and job transfers. On the other hand, with low to moderate overhead costs, none of the other algorithms could match its performance over the range of parameters tested.

# References


[1] R. Agrawal and A. Ezzat. Processor sharing in nest: a network of computer workstations. *Proc. 1st Int'l Conf. on Computer Workstations*, Nov. 1985.

[2] Y. C. Chow and W. H. Kohler. *Computer Performance*, chapter Dynamic Load Balancing in Homogeneous Two-Processor Systems. North-Holland, 1977.

[3] Y. C. Chow and W. H. Kohler. Models for dynamic load balancing in a heterogeneous multiple processor system. *IEEE Trans. Computers*, C-28:354–361, May 1979.

[4] R. Christensen. Entropy minimax multivariate statistical modeling-i: theory. *Int. J. General Systems*, 11:231–277, 1985.

[5] R. Christensen. *General Description.* Volume I of *Entropy Minimax Sourcebook*, Entropy Limited, 1981.

[6] E. de Souza e Silva and M. Gerla. Load balancing in distributed systems with multiple classes and site constraints. *Performance '84*, 17–33, 1984.

[7] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Soft. Engg.*, SE-12(5):662–675, May 1986.

[8] R. G. Gallager. *Information Theory and Reliable Communication.* John Wiley and Sons, Inc., 1968.

[9] K. J. Lee. *Load Balancing in Distributed Computer Systems.* PhD thesis, ECE Dept., University of Massachusetts, February 1987.

[10] M. Livny and M. Melman. Load balancing in homogeneous broadcast distributed systems. *Performance Evaluation Review*, 11(1):47–55, 1982.

[11] R. Mirchandaney. *Adaptive Load Sharing in the Presence of Delays.* PhD thesis, ECE Dept., University of Massachusetts, August 1987.

[12] R. Mirchandaney, D. Towsley, and J. A. Stankovic. Analysis of the effects of delays on load sharing. *IEEE Trans. Computers*, 1987. To Appear.

[13] R. Mirchandaney, D. Towsley, and J. A. Stankovic. Effects of delays on receiver-initiated load sharing policies. *Journal of Parallel and Distributed Computing*, 1987. submitted for review.

[14] J. A. Stankovic. Bayesian decision theory and its application to decentralized control of task scheduling. *IEEE Trans. Computers*, C-34(2):117–130, February 1985.

[15] J. A. Stankovic. Simulations of three adaptive decentralized controlled, job scheduling algorithms. *Computer Networks*, 8(3):199–217, June 1984.

[16] H. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. Soft. Engg.*, SE-3(1), May 1978.

[17] A. Tantawi and D. Towsley. Optimal static load balancing in distributed computer systems. *J. ACM*, 32:445–465, Apr. 1985.

[18] M. Theimer, K. Lantz, and D. Cheriton. Preemptable remote execution facilities for the v-system. *Proceedings of the 10th Symposium on Operating System Principles*, December 1985.

Fig. 1: Effects of Delays (load=0.4)

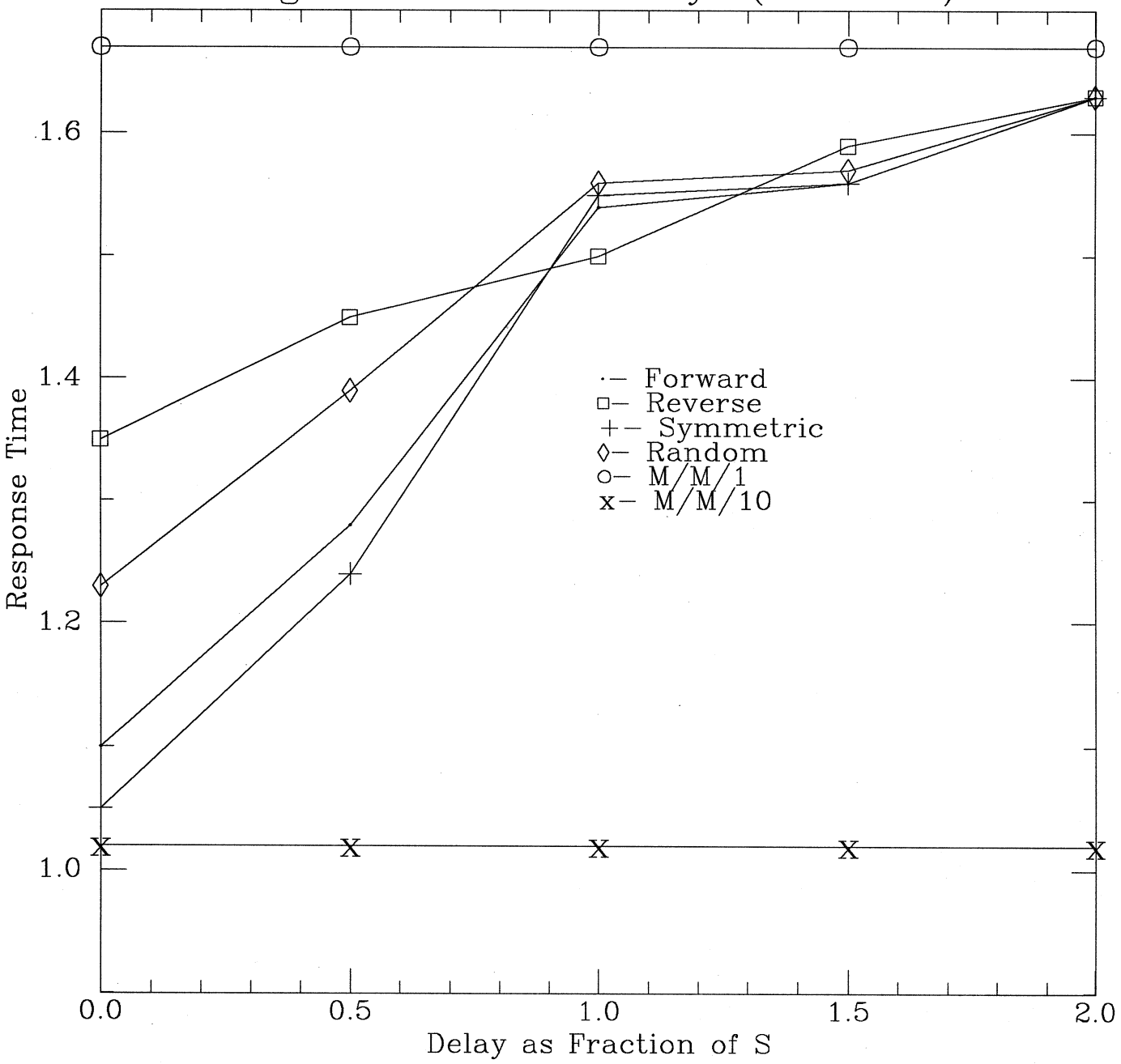

Fig. 2: Effects of Delays (load=0.6)

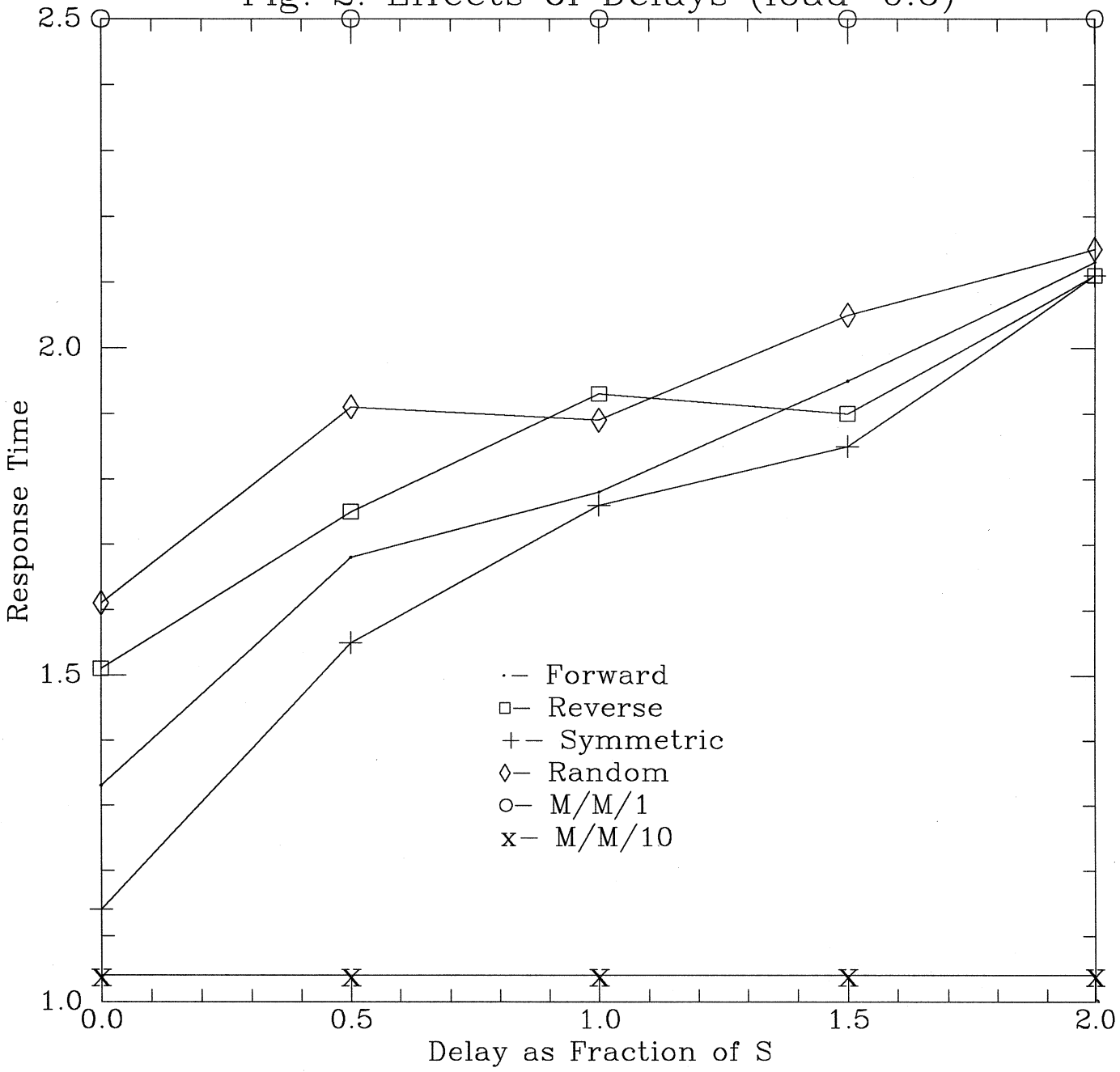

Fig. 3: Effects of Delays (load=0.8)

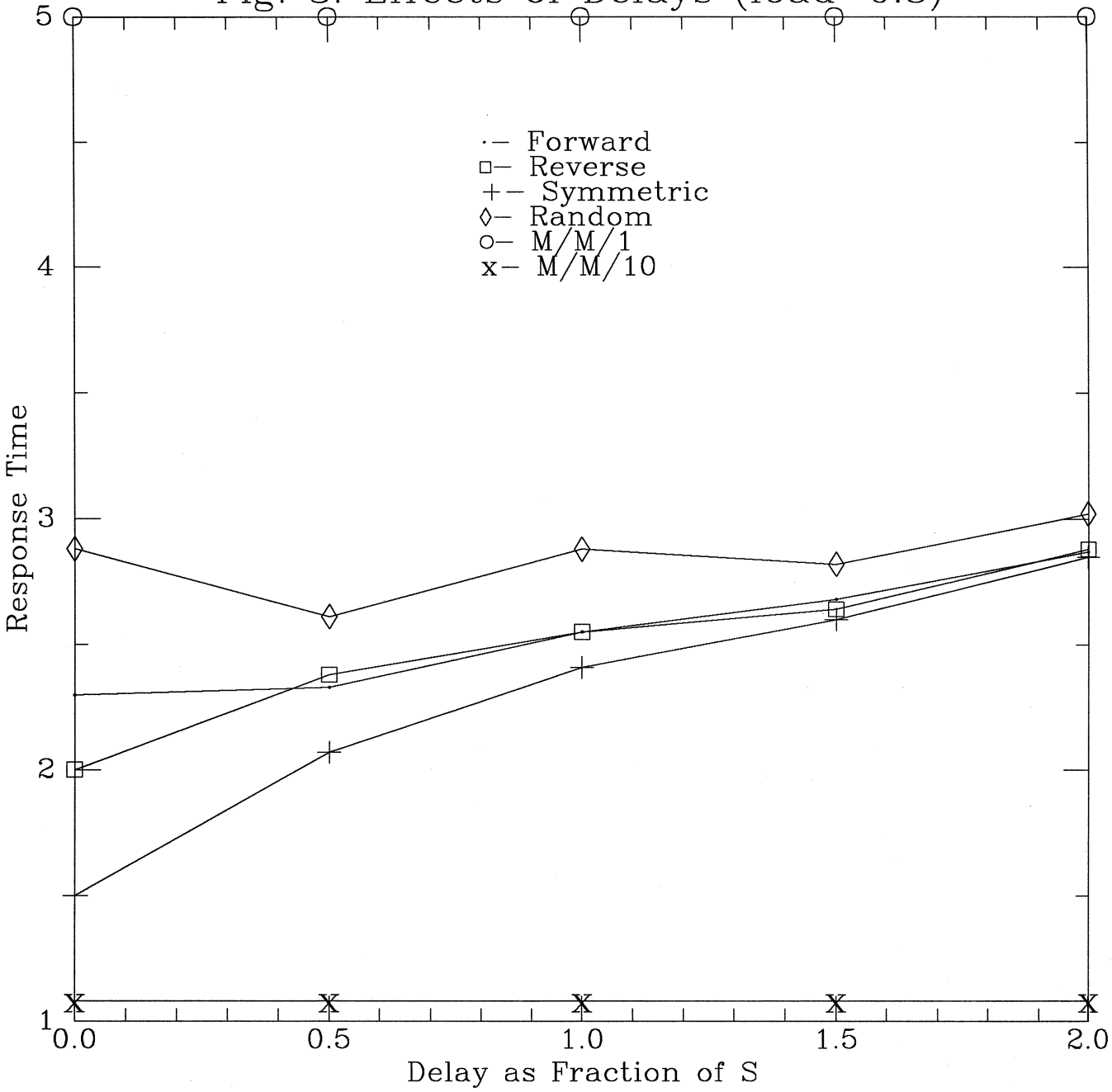

Fig. 4: Effects of Thresholds (Delay=0.5S)

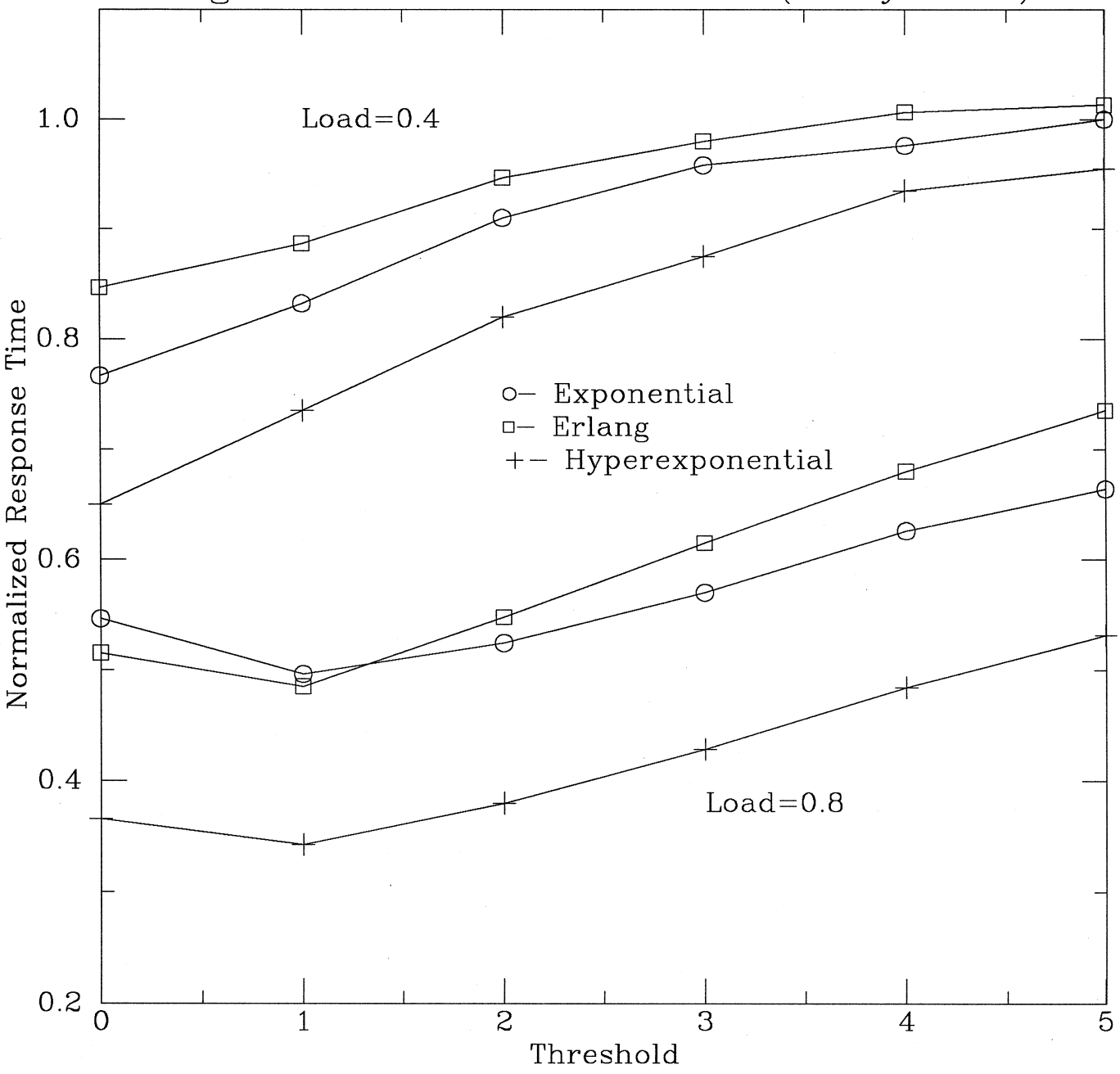

Fig. 5: Effects of Thresholds (Delay=S)

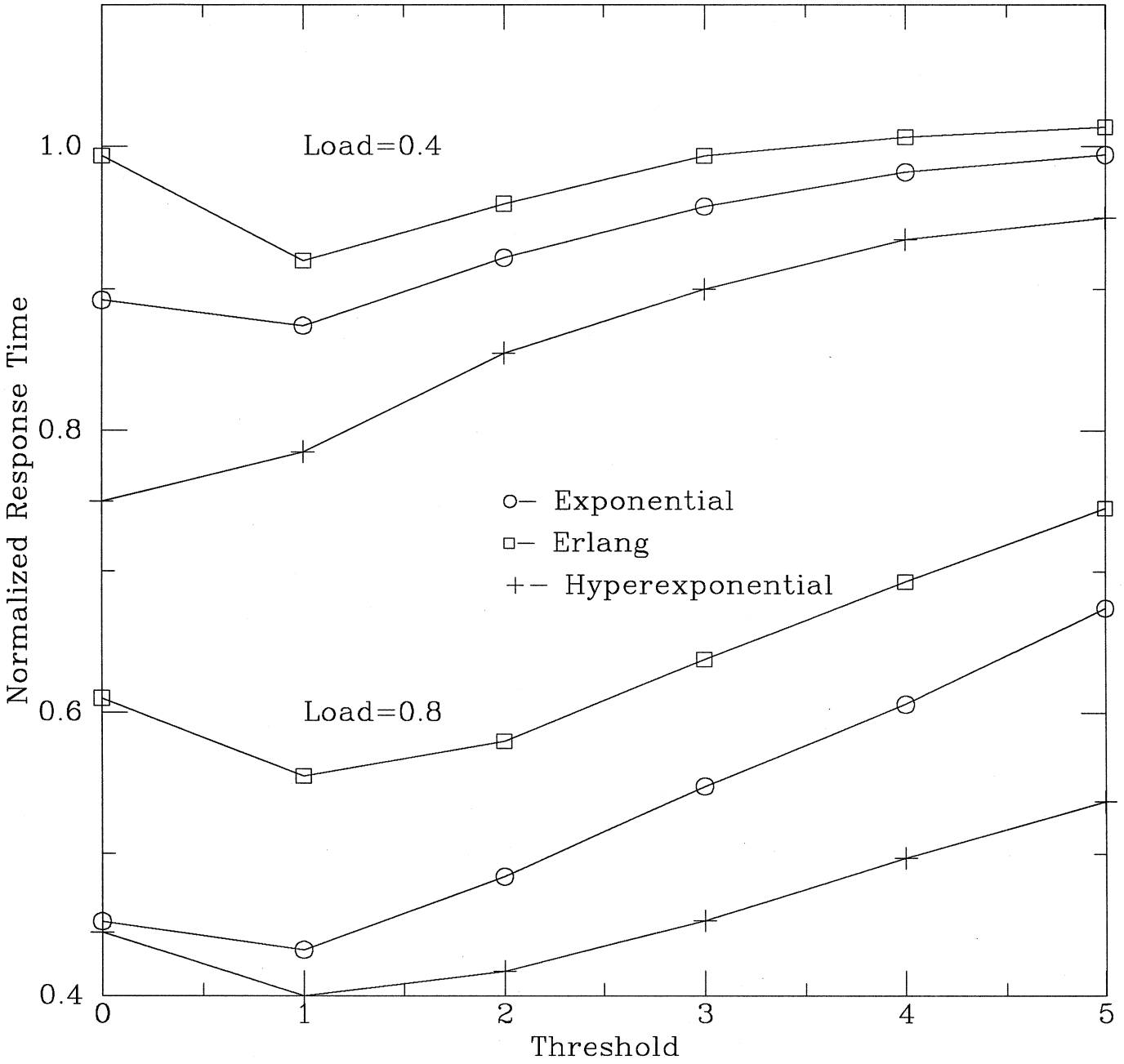

Fig. 6: Effects of Thresholds (Delay=1.5S)

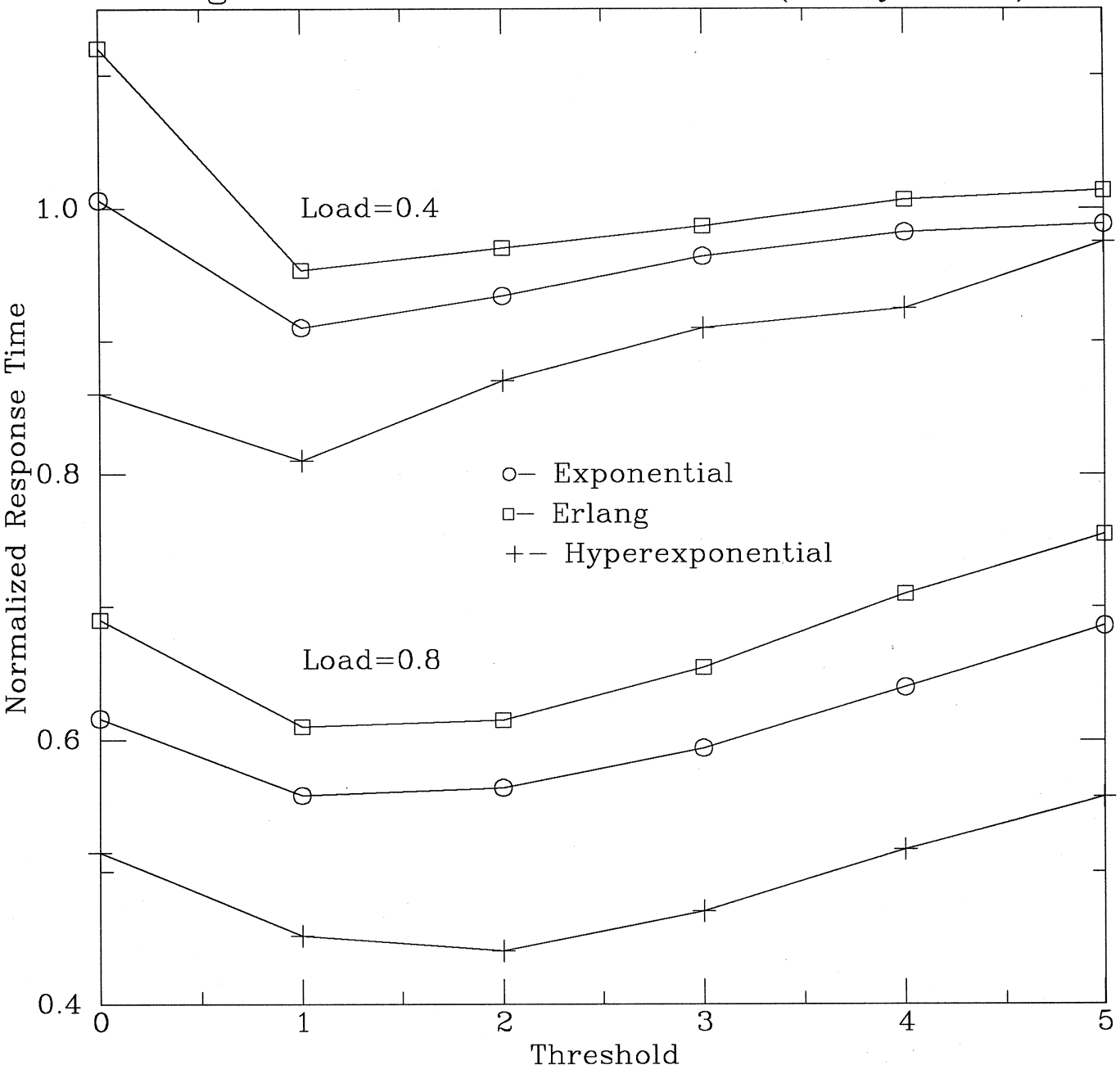

Fig. 7: Effects of Thresholds (Delay=2S)

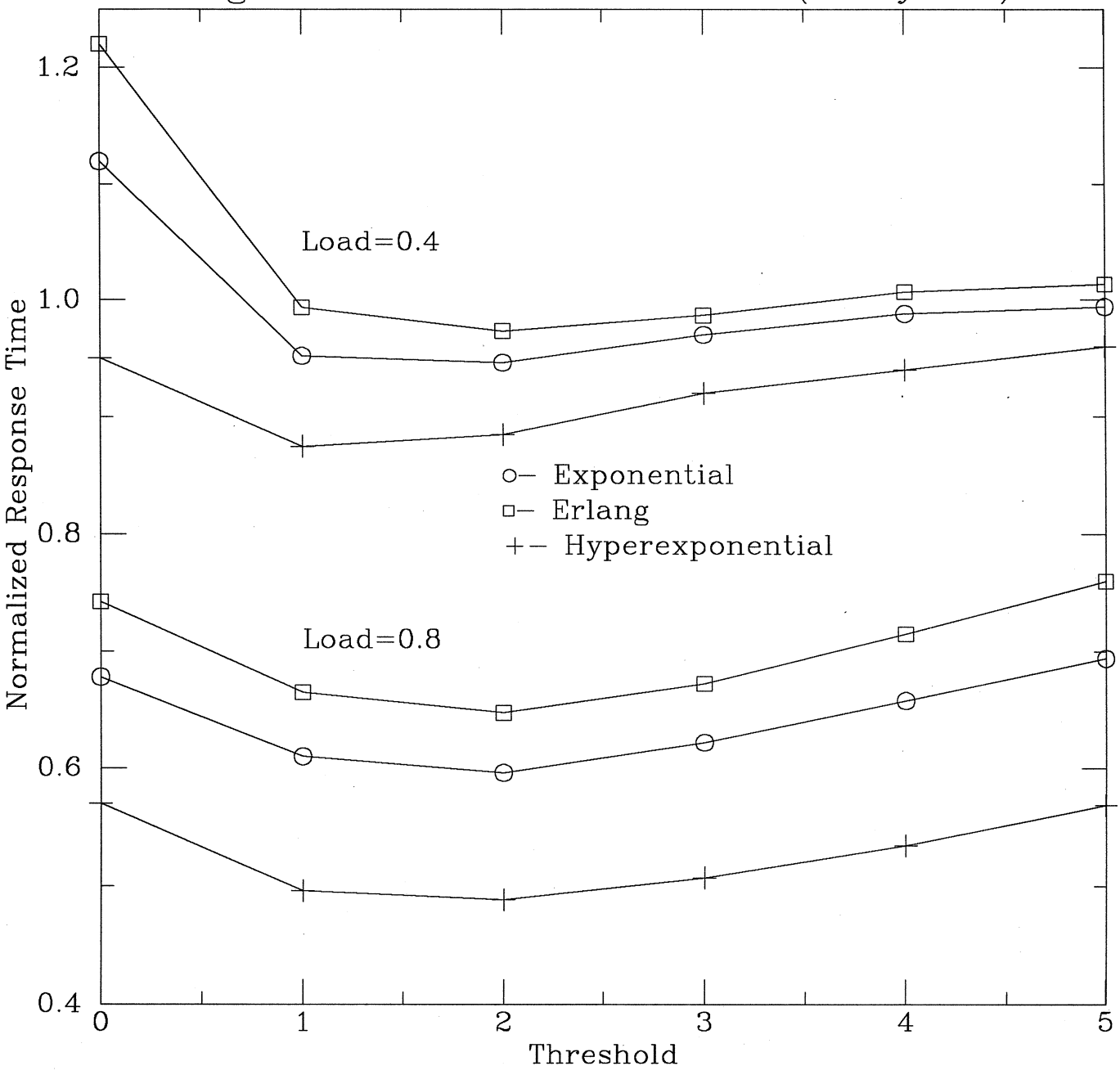

Fig. 8: Effects of Probe Overhead

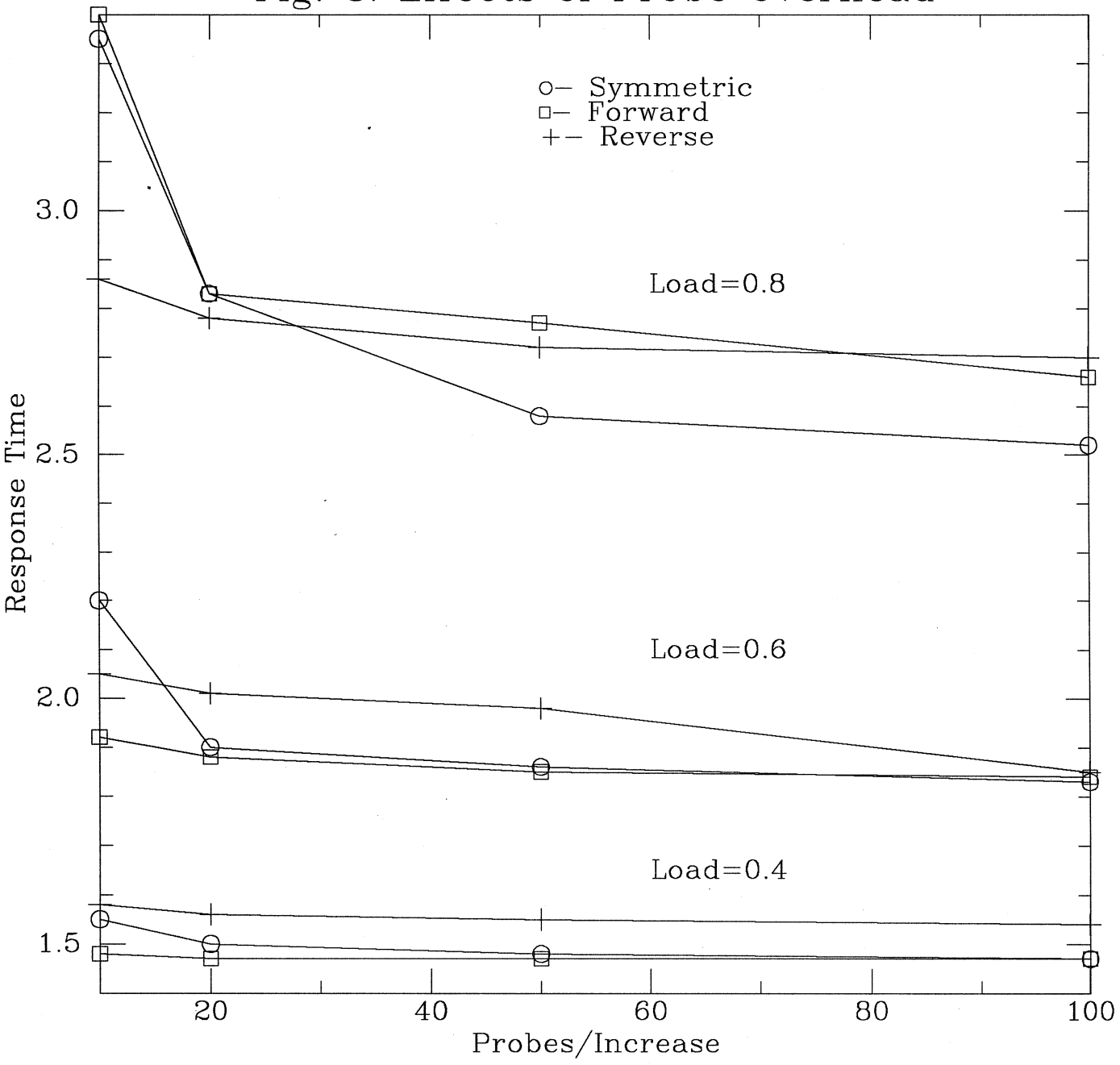

Fig. 9: Effects of Job Transfer Overhead