DISTRIBUTING WORKLOAD AMONG INDEPENDENTLY OWNED PROCESSORS

Luis L. Cova
Rafael Alonso

CS-TR-200-88

December 1988

DISTRIBUTING WORKLOAD AMONG INDEPENDENTLY OWNED PROCESSORS[†]

Luis L. Cova
Rafael Alonso

Department of Computer Science
Princeton University
Princeton, N.J. 08544
(609) 452-5209

ABSTRACT

In many distributed systems it is possible to share the processing capabilities among the nodes. To accomplish this goal, a number of load distribution algorithms have been proposed in the literature. In a network of *independently owned processors* (e.g., a network of workstations), load distribution schemes cannot consider the whole network as one unit and thus cannot try to optimize the overall performance. That is to say that *load balancing* schemes are not appropriate for this type of environment. Instead, the needs of each resource owner have to be considered. Therefore *load sharing* is more appropriate. Load sharing has been accomplished in some systems in an all or nothing fashion, i.e., if a node is idle then it becomes a candidate for executing a remote workload, otherwise it is not. Other attempts have used priority schemes where remote jobs are run with lower scheduling priority than local jobs. These styles of sharing are too restrictive in an environment where most resources are underutilized. Also, they do not scale well as the system load increases. We present a scheme that replaces these sharing approaches with a gradual one, i.e., where each machine in the network determines the amount of sharing it is willing to do. The scheme, called **High-Low**, makes sure that the service provided to local jobs of a lightly loaded node does not deteriorate by more than a predefined amount. It simultaneously helps improve the service at heavily loaded nodes. We empirically compare the different approaches to load distribution and show that it can be effective in a network of workstations.

Index Terms: distributed scheduling, distributed systems, empirical study, load balancing, load sharing, performance evaluation, resource scheduling.

December 13, 1988

DISTRIBUTING WORKLOAD AMONG INDEPENDENTLY OWNED PROCESSORS[†]

Luis L. Cova
Rafael Alonso

Department of Computer Science
Princeton University
Princeton, N.J. 08544
(609) 452-5209

1. Introduction

One of the reasons for the existence of distributed systems is to allow resources to be shared across a network of computers in a user transparent way. One network resource which has received much attention lately has been the processing capability of the nodes. Because each node usually has its own user community and CPU scheduler, an imbalance of the system workload throughout the network can be a common situation. One solution to this imbalance is to allow users at one node to run processes on other nodes in the network. The usual mechanisms for this are remote logins (e.g. *rlogin* (1) [Leffler1984a]), or explicit remote process executions (e.g. *rsh* (1) [Leffler1984a]), but for these mechanisms the selection of the execution site and the control of the remote execution is completely up to the users. It is more desirable to dedicate a system program to the task of sharing the processors in much the same way that memory management software allocates the use of memory. Users can then rely on it to automatically handle remote execution of their jobs in order to take advantage of less loaded processors, thus possibly achieving better average response time.

The **load distribution** problem consist of reallocating the workload in a computer network to achieve better service performance. **Load balancing** has the objective of equalizing the workload at each node while **load sharing** reassigns workloads from heavily loaded nodes to other

---

† This research is supported by New Jersey Governor's Commission Award No. 85-990660-6, and grants from IBM and SRI's Sarnoff Laboratory.

nodes, chosen through some specific criteria (e.g., idle nodes), but not necessarily resulting in an even distribution of the system workload. Most of the literature on load distribution has concentrated on load balancing. This trend can be seen in the many load balancing schemes that have appeared in the published literature: see [Wang1985a] for a proposed taxonomy and a review of the various approaches that have been pursued, or [Zhou1988a] for a comparative performance study of several load balancing policies. Load balancing schemes can be divided in two types: static policies, as in [Ni1985a], which ignore the current system state when making decisions and which usually follow average system behavior, and dynamic policies, as in [Eager1986a], which rely on system state information.

A load distribution scheme is essentially composed of three parts: a transfer policy, a location policy and a load information policy. The first determines if a locally invoked job should be served locally or remotely. If the transfer policy decides to service a job in a remotely, then the location policy determines where the job is going to be executed. For example, a possible location policy would run jobs on the machine which has the lowest load. The last policy is given by the load metric that is used to determine the load in any given machine. For example, a possible load metric could compute the average number of running processes during a certain period of time. Clearly, the selection of a particular load metric depends in the type of jobs submitted to the machine as well as the node's resources and capabilities. Ferrari and Zhou [Ferrari1987a] suggest that an appropriate load metric is a linear combination of resources queue lengths. We define the load of a machine as the value of a given load metric for that machine, and the load of a distributed system as a function of the load of all the nodes in the network. In order for a node to be able to make appropriate load distribution decisions it has to gather information about the load of the system. To do this nodes could exchange their loads.

Load distribution algorithms may comprise a variety of strategies to exchange load information between the nodes of a network. These strategies can range from not exchanging any information to having the nodes possess a common knowledge of the system's load. The former extreme could be implemented by choosing execution sites by means of educated guesses, and

the opposite extreme could be implemented by shared memory, a central controller, or by exchange of frequent messages. There is a tradeoff between these two extreme cases: inaccuracy of the decision made vs. overhead in the decision process. In between these two situations there are many schemes to be explored. The selection of one over another for a particular distributed system is a design issue that affects the service to its users.

Design issues, like the one just mentioned, are influenced by the configuration that the distributed system has. The type of computational environment in which the load distribution strategy is to be implemented has a large impact on the design of the load distribution mechanism. The overall objective of the distributed system has to be taken into account when choosing a strategy because it influences the amount of information to be exchanged among the machines as well as the type of location policy to be used. This relation has been given very little attention by the implementors of load distribution mechanisms.

Distributed systems have some typical configurations. For example, a *pool of processors* is a group of interconnected computers dedicated to servicing equally a group of users. The goal is to provide the user community with improved service. Most of the work described in the load balancing literature has dealt (explicitly or implicitly) with this type of environment ([Stankovic1984a], [Alonso1986a], [Eager1986a], [Zhou1988a]). In this environment all the nodes in the network functionally belong to one organization. The appropriate load balancing scheme for this environment should focus on enhancing the overall system performance which is the system's goal. Examples of this type of system are most of the computer centers in industries and universities.

Another type of distributed system is the *independently owned processors* environment. In this environment each node of the network functionally belongs to a different user, whether that is a single user or a group of users. Instances of this environment are networks of workstations, and networks of inter-departmental machines. Although a load balancing scheme can be used in this type of system, it cannot be treated with the same techniques used with a pool of processors. For this type of environment, *load sharing* is more appropriate.

In networks of independently owned processors, load balancing schemes are not appropriate since one cannot consider the whole network as a single unit and thus cannot try to optimize average response time or system throughput. In any load balancing scheme, heavily loaded nodes will obtain all the benefits while lightly loaded machines will suffer poorer response time than in a stand-alone configuration. Users of a frequently heavily loaded machine will cheer for a load balancing scheme while users of mostly underutilized ones will strongly oppose participation in such a scheme. What is desirable is a fair strategy that will improve response time to the former without unduly affecting the latter.

In some systems this issue has been resolved by implementing an "all or nothing" strategy. If a machine is completely idle then it becomes a candidate for executing a remote workload. If a machine is being used, even if it is underutilized, then no remote workload is allowed. Basically, any machine can take over an idle one in a master-slave relation, but as soon as the owner of the idle machine uses it (even slightly) all the remote jobs are either put in the background (run with low priority) [Hagmann1986a], moved back to their originating node [Theimer1985a], moved to another idle machine [Litzkow1987a] or just killed (abnormally terminated) [Nichols1987a]. While all of these techniques guarantee the ownership of resources to the owner of an idle machine, they do not assure any performance improvement to the remote jobs the idle node may be servicing. It is desirable to have a more gradual style of sharing that would attempt to guarantee processor performance to node owners as well as offer some help to the remote jobs that may have been submitted to a node.

Our purpose is to adapt the results obtained for the load balancing problem to load sharing among independently owned processors. To do this we suggest the use of a fourth policy: the **acceptance** policy. The acceptance policy reflects the disposition of a node owner to accept a certain level of remote jobs to be serviced by his or her machine. In other words, our interest is to provide the owners with control over their machines independently of the load distribution scheme being used.

Throughout this paper, whenever we refer to the load of a machine we mean a consistent load metric that characterizes the usage of that machine. We will be concerned only with the initial placement problem, i.e., where in the network a job should be run, and we will not consider the migration of jobs once they have started running in a node.

In the following section we will present the definition and description of the High-Low scheme. In the same section we will present an analytical model to estimate the values of the High-mark and Low-mark parameters. In Section 3 we will describe a load balancing algorithm based on running jobs at the least loaded machine in the network. We will empirical compare this scheme to High-Low and to the case when no load distribution takes place. In Section 4 we will present other acceptance schemes used for load sharing and we will empirically compare them to High-Low. We conclude by summarizing our findings and by suggesting directions for further research.

2. The High-Low Scheme

Computers participating in a system where load sharing takes place may be viewed as being at any one time either sources of jobs or servers of jobs. When a machine is viewed as a source of jobs, a machine should only try to execute remote jobs if transferring some of them to another node will greatly improve the performance of the rest of its local jobs. This observation parallels the usual banking practice of borrowing only when necessary. On the other hand, when a computer is viewed as a server of jobs it should only accept remote jobs if its load is such that the added workload of processing these incoming jobs does not significantly affect the service to the local ones. This approach mirrors the sound banking practice of only lending excess funds†. These two notions can be adapted to a load sharing environment via two policies that we denote by **High-mark** and **Low-mark**.

† It should be pointed out that the banking analogy does not hold completely. In contrast to the banking situation, borrower nodes need not return their borrowed cycles and lender nodes may not receive back their lent cycles.

The High-mark policy behaves as follows: each time the execution of a new job is requested at a node the load of the machine is compared against its High-mark value. If the former is greater than the latter then the load sharing mechanism tries to execute the job in a remote host. Otherwise the job is processed locally. Thus, High-mark sets a lower level on the load a machine must have before it begins to transfer jobs to other hosts. Its purpose is to try to reduce processing overhead by load sharing only when the workload of the machine degrades its service dramatically.

The Low-mark policy sets a ceiling on the load a computer may have and still accept incoming remote jobs for service. Its purpose is to be able to handle these incoming jobs while the service to the local ones is not significantly affected. Low-mark works as follows: whenever a request to execute a remote job arrives at a machine, the processor checks if its load is less than its Low-mark value. If so, then the request is accepted and the job is processed locally. Otherwise the request is rejected.

Clearly, High-mark and Low-mark may be implemented together. We refer to this combined policy as **High-Low**. At this point we should contrast this work with that of Eager et al. [Eager1986a], who also proposed threshold policies for load sharing. Their analysis used the same threshold value for deciding when to offload work to other nodes (transfer policy) and for deciding where to run a remote job (location policy). In our scheme, the High-mark plays the role of the transfer policy, while the location policy can be chosen depending on the situation. Low-mark represents the acceptance policy. High-mark and Low-mark would be used in addition to location policies to control when a remote interaction should take place.

2.1. Description

Figure 1 presents the algorithm for the High-Low scheme. In our first implementation we chose a random allocation of jobs as the location policy, although it is not a particularly good choice. Under this location policy, the executing node for a job is selected at random and the job is transferred there. No exchange of information is done between the machines in the network. It

is simple to implement and no system state is available to the nodes. [Eager1986a] and [Zhou1988a] showed that randomly selecting where to run a job reduces the system's average response time, but it is very unstable, i.e., its improvements depends on the system's workload. As will be seen later, our scheme reduces the instability of the random allocation policy by restricting the interchange of jobs between the nodes. We also implemented another version of High-Low using a least loaded node selection as location policy.

The salient feature of the High-Low scheme is that two different thresholds are used to decide if a job is to be run remotely. This allows a computer to play multiple roles depending on the values that High-mark and Low-mark take. For example, Figure 2a shows that if the High-mark value is greater than the Low-mark value then the space of possible load values that a machine can have is divided into three regions:

1) **overloaded** (above the High-mark and Low-mark values);

2) **normal** (above the Low-mark value and below the High-mark value);

3) **underloaded** (below both values).

When the load of a machine is in the overloaded region, new local jobs are sent to be run remotely and remote execution requests are rejected. In the normal region, new local jobs run locally and remote execution requests are rejected. In the underloaded region, new local jobs run locally and remote execution requests are accepted.

Most load sharing algorithms use a single threshold (typically the "average" load of all the network processors), and thus only have overloaded and underloaded regions (Figure 2b). High-Low defines a third region (normal) by its use of two different thresholds. This normal region guarantees a predefined level of performance to the node owners. It may account for the overhead that the load sharing scheme incurs in transferring and receiving a remote job, and for the level of service that the owner expects. A job will not be transferred to another node unless it is worthwhile and a remote job will not be accepted unless there is enough excess capacity to handle it.

Notice in Figure 2c that if the Low-mark value is allowed to be greater than the High-mark value, then a fourth region could be recognized: the **undesirable** (above the High-mark value and below the Low-mark value). In the undesirable region the machine would send its new local jobs to remote processors while accepting remote jobs to be executed locally. This is a form of processor thrashing. To avoid this anomaly, the High-mark value should always be greater than or equal to the Low-mark value. However, if High-mark and Low-mark have the same value (Figure 2b) then the implementation of High-Low behaves as a load distribution algorithm that uses a single threshold.

Figure 3 shows how particular settings of the High-mark and Low-mark parameters correspond to particular modes of operations of a computer. For example, by setting both the High-mark and the Low-mark to 0 (or the lowest possible value), a computer acts as a job dispatcher (Figure 3a). The computer behaves as if it were always overloaded: it places all of its new local jobs in any available remote machine. This setting could be used to distribute jobs to a pool of processors. If instead, both parameters are set to the maximum possible load value (Figure 3b), then the machine would act as if it were always underloaded, i.e., it would accept any remote process for execution. Thus, the computer is behaving as a process server.

It could be possible to vary the High-mark and Low-mark parameters dynamically to allow the computers to have different modes of participation in the load sharing scheme. A system process could set these parameters depending on the number of users, or user processes, in the computer. For example, when the last user signs off, the High-mark and Low-mark values could be set to leave the machine in the process server mode (Figure 3b). As soon as a user signs in, these parameters could be set back to leave the computer in its normal operational mode. If a computer is needed exclusively by its owner then the High-mark and Low-mark parameters could be set to the stand-alone mode (Figure 3c). The attraction of this scheme is that different modes of operation can be easily implemented by dynamically setting the High-mark and Low-mark parameters.

An important observation is that the degree of participation of each computer in this load sharing scheme is completely distributed. It does not depend on any global information or central controller, just on the node's local use and purpose.

Finally, since the load information of a machine represents the available resources in that machine and Low-mark represents the amount of resources that the node's owner is willing to lend, in schemes that require processors to make known their load, the Low-mark value could be included with the machine's load information. In this way other nodes would know in advance the available resources in the network and could plan to use them.

2.2. An Analytical Model for setting High-mark and Low-mark

Choosing appropriate values for High-mark and Low-mark is not a simple task. An automatic fine tuning mechanism together with specifications submitted by machine owners could be used to obtain the best results. For example, a user could specify that he will allow his machine to process remote jobs if the average response time for his jobs does not deteriorate by more than 10% of the stand-alone time. Selecting the High-mark and Low-mark values is a continuing area of our research. Below, we use a simple analytical model to estimate the High-mark and Low-mark values.

We will model a node as a M/M/1 queue [Kleinrock1976a].

Let λ = the number of jobs arriving at the node per time unit, $\frac{1}{\mu}$ = the number of jobs that can be processed by the node per time unit. Then we know that the average response time (T) of a job in that system will be $\frac{\frac{1}{\mu}}{1-\rho}$ (where ρ is the processor utilization, $\rho = \frac{\lambda}{\mu}$) and on average there will be $\bar{N} = \frac{\rho}{1-\rho}$ jobs in the machine.

Suppose that the users of that machine believe that the normal range of response time for local jobs should be $T \pm \Delta$. Thus, the High-mark should be a load average of N_{high}, such that with N_{high} jobs in the system, a job obtains a response time of $T + \Delta$, and the Low-mark should be set to a load average N_{low}, such that having N_{low} jobs running concurrently will lead to a response

tome of $T - \Delta$.

Thus, the new T (T') will be :

$$T' = T \pm \Delta = \frac{\frac{1}{\mu}}{1-\rho'}$$

which implies,

$$\rho' = 1 - \frac{\frac{1}{\mu}}{T \pm \Delta} = 1 - \frac{1-\rho}{1 \pm (1-\rho)\,\mu\,\Delta}$$

and hence

$$N_{high} = \frac{\rho'}{1-\rho'} = \frac{1+(1-\rho)\,\mu\,\Delta - 1 + \rho}{1-\rho} = \frac{(1-\rho)\,\mu\,\Delta + \rho}{1-\rho} = \bar{N} + \mu\Delta$$

and similarly,

$$N_{low} = \bar{N} - \mu\Delta$$

For example, consider a system with an arrival rate of 8 jobs/second with a CPU that can process at most 10 jobs/second. Then, $\rho = 0.8$ and on average $\bar{N} = 4$ jobs, and $T = 0.5$ second. If $\Delta = 10\%$ we can compute that the High-mark is $4 + 10(0.1) = 5$, and the Low-mark is $4 - 10(0.1) = 3$. This means that the machine, to guarantee the specify response time ($T \pm \Delta$), should try to execute some of its jobs remotely when its workload is greater than 5 jobs (*load average* $> N_{high}$) and that it could serve jobs from other nodes when its workload is less than 3 jobs (*load average* $< N_{low}$).

3. High-Low, lsh and no-ld

In [Alonso1986a] a load balancing scheme (called **lsh**) was developed based on broadcasting local system state to all the nodes in a local-area network (LAN) and on transferring jobs to the least loaded node. Each node reaches load balancing decisions in a decentralized fashion,

i.e., without the existence of a central controller. The purpose of the prototype was to demonstrate that sizable overall system performance gains could be achieved using a simple load balancing mechanism on top of an existing system with small overhead and making very few changes in the underlying software. It was noticed that having accurate information about the entire system was expensive because processing broadcast messages from other nodes takes a substantial amount of CPU cycles. There is a tradeoff between the broadcasting interval and the processing overhead which directly affects the accuracy of the information on which a machine has to base its locality decision. In a follow-up study [Alonso1986b], this tradeoff was discussed and the issues involved in evaluating load metrics and decision policies were described.

Lsh was revised and improved to take care of obvious flaws that a simple "least loaded" scheme has. These flaws are the swamping and drought effects. These effects are produced by the same factor: outdated system state information due to update interval and communication delays. In [Williams1983a] the importance of this factor is recognized in the design of load balancing strategies.

In the swamping effect many jobs are sent to one machine (the least loaded at that moment) before it can broadcast its new load. This occurs because several machines may choose to transfer jobs to the least loaded site within the same small interval of time, before new state information from the least loaded machine is broadcast. Therefore, the response time of these transferred jobs may even be greater than if they had been processed in their originating nodes.

In the drought effect, truly least loaded nodes do not receive remote jobs. This happens because the moment a machine gets less loaded or even idle is not synchronized with its broadcast interval. A node may be the least loaded site in the network, but until it broadcasts its new state, no other node will know it.

To correct the above described anomalies two policies were incorporated into lsh: required load difference and implied load. The required load difference limits a machine to send jobs to a remote node only if the difference between its load and the load of the remote machine is greater than some specific amount. This would reduce remote execution overhead. With implied load

the system load information kept at a machine is updated each time a local job is migrated to another node, i.e., when a machine transfers a job to another node, it adds a certain amount to its information about the receiving node. This is done to compensate for the added workload at the remote site, until an updated load message is received. In this way the sending machine has more accurate information when making the next load balancing decision.

As stated in the introduction, the objective of load balancing schemes is to equally distribute the workload among all the participating nodes. We will show empirical measurements of lsh to compare it against different implementations of High-Low. Also for comparison purposes, we will show performance measurements for the case when no load distribution is done.

3.1. Experiments and Results

The system we used for our experiments uses a network of workstations. Our environment consists of four identical single-processor machines (SUN 2[†]) connected by an Ethernet [Metcalfe1976a] and using a fifth machine as a file server. The load metric we used for our experiments is the UNIX[‡] 4.2 BSD "load average" metric provided by the *uptime* (1) command [Leffler1984a] and defined as the exponentially smoothed average number of jobs in the run queue over the last 1, 5 and 15 minutes. In our experiments we use the average over the last minute. This is the load metric we used for all the experiments in this report. Our experiments also used numbers related to this "load average" metric as High-mark and Low-mark values.

Using these facilities we first implemented the High-Low scheme using a random allocation of jobs as location policy. We ran several tests with our implementation and compared the obtained results against the situation when there is no load distribution and with the lsh implementation (labeled respectively **no-ld** and **lsh** in our figures). In our experiments, each user is simulated by a script and the time it takes to complete is what we define as response time. The

† SUN 2 is a trademark of Sun Microsystems, Inc.

‡ UNIX is a trademark of AT&T Bell Laboratories.

script consist of repeated cycles of editing, compiling and running a C program. The C program performs several arithmetic operations.

Emphasis was not only on the average response time of the system (i.e., of all the nodes), but also on the average response time of the jobs at each individual node. This last measurement gives an idea of the changes in local service time when a machine participates in a load sharing scheme.

Figures 4 through 6 show the average response time of the High-Low scheme with a fixed High-mark value (1.75) and several Low-mark values. The High-mark value was selected after running several experiments with just the High-mark parameter [Cova1988a]†. Each figure represents a system with a different system load. Each node in the network has a particular number of users. For example, the distribution "5,1,1,1" represents the number of users in the system, i.e., five users in one computer and a single user in each of the other nodes. This distribution, "5,1,1,1", represents a low system load, "5,5,1,1" represents a medium one and "5,5,5,1" represents a high system load.

The abscissa depicts a range of increasing Low-mark values and two special values one for no-ld and one for lsh. These values represent the amount of CPU cycles that each particular node dedicates to service remote jobs: from no sharing to full sharing. The labels of the ordinate denote the average response time of jobs submitted by the local users at each node. In each figure there are three bars labeled "5","avg", and "1". The "5" and "1" bars represents the average response time perceived by the users submitting jobs at the machines with five users and a single user, respectively. The "avg" bar is the average response time of the entire system (i.e., of all the jobs).

The first noticeable result from all these figures is that no user perceives an average response time for its jobs close to the average response time of the system. This last

† In [Cova1988a], after testing High-mark values ranging from 0.5 up to 3.25, we concluded that using a High-mark value that represents the average load of a user is a good threshold for deciding if a local invoked job should be run locally or remotely.

measurement is what is usually reported in load balancing studies.

Figure 4 also shows that even with a small willingness to share (represented by a Low-mark value of 0.4), there is a substantial improvement in the performance of the heavily loaded machines while the lightly loaded ones are not significantly penalized. Also, as the Low-mark value increases (more sharing is allowed) the response times of all the machines tend to show a balanced effect, i.e., High-Low's performance is similar to lsh's.

Figures 5 and 6 show that even with increased system workload it is still possible to obtain performance improvements at the heavily loaded nodes without significantly affecting the service at the lightly loaded ones. This is not true when there is complete sharing, as is the case of load balancing schemes. Consider the behavior of lsh in the same figures. It does not scale well as the load of the system increases. Also, notice that in these figures the average response time (avg.) does not uniformly decrease as the Low-mark increases (Low-mark 1.4 in Figure 5 and Low-mark 1.2 in Figure 6). These "bumps" are produced by the random nature of the job's execution location selection. As it can be seen, even with this anomalous behavior the response time of all the heavily loaded machines is still lower than in the no-ld case and, in some cases, lower that the lsh scheme.

Figure 7 presents a summary of the results for a High-Low implementation that uses a "least-loaded" allocation of jobs as location policy. In this figure the abscissa represents the system workload, as explained before. For each abscissa label, there are three columns: one for each type of machine in the experimental system (heavily loaded ones, with five users, and lightly loaded ones, with a single user), and one for the average response time of the system. In each column the response time for each Low-mark value, no-ld and lsh is represented. The High-mark value is fixed to the same value as before (1.75).

From Figure 7 we can see the incremental changes in performance as we change the Low-mark value. It is clear from this Figure that even when there is a lot of activity in the system (user distribution "5,5,5,1"), some improvement in response is achieved by sharing resources with High-Low. Also, the higher the value of the Low-mark parameter, the more the response

time of heavily loaded machines improves, but the more the response time of lightly loaded machines degrades. Notice the excessive penalty that lightly loaded nodes are paying when there is complete sharing (lsh case). This behavior gave us the insight that load distribution algorithms in order to support autonomy should not behave in a binary fashion (share all or share nothing). Instead they should be gradual as is High-Low. In [Alonso1988a] we extended this idea to resource sharing in distributed environments.

Also note that, as with the first implementation, there are some High-Low set-ups - under medium and high system load - that have lower average response time than lsh. In the next section will discuss further this observation.

For the rest of the figures in this paper we will use one of the formats just described for the figures in this section.

4. High-Low, All-or-Nothing, and Priority

Informally, there have been other acceptance policies discussed and used in the load sharing literature. First, is the extensively used **All-or-Nothing** scheme where idle processors are used to service remote jobs until they are claimed by their owners ([Nichols1987a], [Litzkow1987a], [Theimer1985a], [Agrawal1987a], [Mutka1987a], etc.). Second, is the **Priority** acceptance policy where remote jobs are accepted at a node with lower scheduling priority than local jobs ([Hagmann1986a], [Leland1986a], etc.). These methods clearly guarantee ownership of resources to the machine's owners. We will empirically compare them against our High-Low scheme by using a synthetic workload.

We decided that to fairly compare the different acceptance policies we had to use the same schemes for the other policies involved in load distribution algorithms, i.e., the information, transfer and location policies and to have more general results, we decided to use two different sets of information, transfer and location policies. It was also important to have a comparable overhead cost for the implementation of each acceptance scheme. To ensure this we chose to emulate the different acceptance policies by using our High-Low implementation.

One set of policies consisted of no information exchange among the nodes, a threshold transfer policy and a random selection of execution site as the location policy [Alonso1988b]. The other set consisted of a periodic exchange of load average information among the nodes as the information policy, running each job in the least loaded node as the location policy and using a required load difference between the least loaded node and the originating node as the transfer policy [Alonso1986a]. We denote the implementation of the former set by **random** and the implementation of the latter set by **lsh**† (the lsh name is used because the corresponding set of policies is based on the lsh scheme described in Section 3).

4.1. Description of the emulated acceptance policies

As explained in Section 2, High-Low can be used to emulate different modes of node operations. By slightly modifying our High-Low implementation we were able to emulate the All-or-Nothing policy and the Priority policy.

The All-or-Nothing policy was emulated by using High-Low with a High-mark set to the lowest possible value (0.01) for the lsh implementation and a median value (1.75) for the random implementation. The Low-mark was set to the lowest possible value (0.01) for both sets. The High-mark setting acts as the threshold for the transfer policy in the random set. In the lsh set it makes the scheme look for the least loaded node in the network to offload a remote job. The Low-mark guarantees that remote jobs only get accepted if the load average of the node is almost zero, i.e., if the machine is idle.

The Priority policy was emulated by running the process that handles remote executions (the lshd daemon [Alonso1986a]), with the lowest scheduling priority possible (a nice(1) value of +20 [Leffler1984a]). This process behaves as follows: when a remote job is accepted for execution at a node, it will instantiate (fork) a child process that will take care of getting the

† There are many parameters involved in each set of policies. We have done a limited sensitivity analysis on these parameters, in particular for the lsh set, in [Alonso1986a] and [Cova1988a].

environment information of the job and running a local instance of it. This child process will have the same scheduling priority as its father , thus the remote job will run with low scheduling priority, too. For both sets, High-mark was set to the same value than for the All-or-Nothing emulation. The Low-mark was set to a large value (100) for both implementations to assure that every remote request would be accepted.

4.2. Experiments and Results

The experiments we ran were similar to the ones described in Section 3. Along with the two previously mentioned acceptance policies, we gathered results from three different instances of the High-Low algorithm:

- A configuration where all the nodes have their High-mark set to the average load of the network (1.75), and the Low-mark set to a very small value (0.4), thus allowing little remote workload service (labeled "0.4/1.75" in our figures).
- A configuration where heavily used nodes ,i.e., nodes with 5 users, allow no remote workload service (Low-mark = 0.01) and have their High-mark set to the average system's load (1.75); the lightly used nodes ,i.e., nodes with a single user, allow some remote workload service (low-mark = 0.6), and their High-mark is set higher than the system's average (2.0). In this way these nodes will not try to offload their own workload during transient conditions due to remote workload service (labeled "0.01/1.5-0.6/2" in our figures).
- A configuration where heavily used nodes allow very little remote workload service (Low-mark = 0.4) and have their High-mark set to a much higher value than the system's average (2.4). In this way they will only try to offload their workload when their local service has greatly degraded. The lightly used nodes allow a considerable remote workload service (Low-mark = 0.8) and only will try to offload their own workload when their load is well above the average (High-mark = 3.0) (labeled "0.4/2.4-0.8/3" in our figures).

We also gathered results for the cases where no acceptance policy is used, i.e, the random and lsh sets of policies by themselves, as well as for the case when there is no load sharing (no-

ld). Thus, a total of seven load sharing situations were tested per set of policies.

In Figures 8 through 10 we present the response time of the different acceptance policies using the lsh set of policies† and under different system loads (user distributions). The first noticeable result is that for all the system loads the performance of the All-or-Nothing policy can always be improved, i.e., the All-or-Nothing policy is only better than the no load distribution case (labeled no-ld in the figures), but in some cases, when the system load is extremely high (Figure 10), it can even be worse than this case. The All-or-Nothing policy has worked well for many researchers because their systems load is constantly low [Mutka1987b]. There is always a high probability of remote processing availability due to idle processors in the network. As can be seen in Figures 8 through 10, there are other policies that have the same characteristics than the All-or-Nothing policy, but do not have this observed anomaly. The other policies behave well under increasing system load and even do better under low system load. The only observation in favor of this policy is that its standard deviation (Figure 11) tends to be lower than other policies, in particular for medium and high system load. This happens because less interaction among nodes occurs as load increases.

In brief, our results suggest that the All-or-Nothing acceptance policy can always be improved, no matter the system load. It does not have any feature of merit except that it is the obvious way (and simplest to implement) to guarantee ownership of resources.

Examining now the results from the Priority scheme, we notice that it achieves considerable performance improvement for the heavily loaded processors, but it also penalizes more the lightly loaded nodes in comparison to other acceptance schemes. As the system load increases the average response time of the lightly loaded nodes degrades. Also, a higher variability (Figure 11) is present for both type of nodes. This anomalous behavior has to do with the load information policy used for lsh. Our load average measure does not distinguish between local jobs

† Although we have collected results using the random set, we will not present the corresponding figures in this report. We do this to limit the number of figures in this presentation, even though they support the conclusions drawn from the lsh set.

and remote jobs, i.e., it expresses the actual load of a node regardless of what type of job is producing it. Thus a lightly loaded node may get a number of remote jobs to service (placed in the background) which increases its load average value. The next local job that arrives will see a high load average value which will induce the load sharing software to place such a local job in a remote processor *with low scheduling priority*. This is reflected in a poorer response time than if it was run locally. It is a form of thrashing. Therefore, the lightly loaded nodes are effectively being penalized.

Researchers using this policy explicitly allocate jobs throughout the network [Hagmann1986a], [Mutka1987a]. Users of lightly loaded nodes do not use the load sharing mechanisms, while users of heavily loaded nodes explicitly request that part of their workload be placed at remote machines. This approach to using the load sharing software goes against our assumption that such mechanisms should transparently take care of remote executions on behalf of inexperienced users (as stated in our introduction).

One way to correct this anomaly in the Priority policy is to be able to distinguish the load generated by local jobs from the load average of any particular node. In other words, the load sharing mechanism could use two load measurements: one for foreground or local jobs and one for the entire workload. Clearly, this improved Priority policy relies on the use of more information to based its load balancing decisions, i.e., a different information policy that the ones we have been using. Therefore it is not reasonable to compare it with the other acceptance policies in this report.

The Priority scheme has the largest standard deviation for all system loads (Figure 11). The variability increases in direct response to the increase in system load. This is an undesirable behavior because users of the load sharing software cannot estimate the response time for their jobs.

With respect to the load sharing algorithms not using an acceptance policy (lsh), we notice that the lightly loaded nodes are heavily penalized. The system improvement comes from equalizing the system workload among all the nodes. The degradation is more noticeable when the

system load is high. As we have already noted in [Cova1988a] this scenario is acceptable if the entire network belongs to the whole user community, i.e., the system is being shared equally by all its users. This situation is not appropriate when each node is privately owned by different users because the owners loose control over their resources. Again, guaranteeing resource ownership is the motivation behind the acceptance policy.

Let us now consider the High-Low results. Since High-Low allows a gradual sharing of the resources, it offers a marked improvement over the All-or-Nothing policy and the no-ld case. It also provides better control of the local resources than the Priority scheme, i.e., a lightly loaded node participates in the load sharing mechanism only to the extent that its owner allows. Also, because each node pledges a portion of its resources to service remote jobs, any remote job have some assurances over the quality of service it is going to receive. This property is not present in any of the other acceptance policies.

A final observation is that the standard deviation of High-Low tends to decrease as the system load increases, differently from Priority and similarly to All-or-Nothing. As with All-or-Nothing this is due to the fact that less remote processing occurs as the system load increases, which is desirable to avoid processor thrashing.

5. Conclusions

We have discussed the independently owned processors environment, a type of distributed system where sharing the processing capabilities among the nodes improves performance. We have argued that load balancing algorithms are not appropriate for this type of environment because of its characteristics of autonomy and local ownership of resources at each node.

The sharing scheme we have presented, called High-Low, replaces the notion of stealing CPU cycles with the notions of lending and borrowing CPU cycles. Workstation owners do not have to be concern with their resources being abused. The degree of participation of each computer in this load sharing scheme is completely distributed. It does not depend on any global information or central controller, just on the node's local use and purpose.

The All-or-Nothing and Priority acceptance schemes of load sharing are too restrictive in an environment where most resources are underutilized. Also, they do not scale well as the system load increases. Instead, by allowing a moderated sharing among the nodes, good performance is guaranteed to the node owners and to the remote jobs.

Our results suggest that the All-or-Nothing acceptance policy can always be improved, no matter the system load. It does not have any feature of merit except that it is the obvious way (and simplest to implement) to guarantee ownership of resources.

High-Low has the desirable characteristic that an All-or-Nothing scheme has, i.e., it guarantees to the owners of lightly loaded machines that their local resources will not be abused by remote jobs. This is achieved by fixing the maximum degradation that a user of a lightly loaded node might perceive. At the same time, it avoids the anomaly of having poorer performance for the heavily loaded machines as the load of the network increases (when it should have an opposite behavior since it is in this situation that improvement is most needed). The High-Low scheme improves the performance of heavily loaded nodes in a greater amount than All-or-Nothing or Priority, and is close to the performance of the load balancing schemes tested (lsh and random).

A further point is that the capability to migrate jobs [Alonso1988c] after they have started executing in a machine may be desirable for our system. For example, a machine may receive too many remote jobs which can suddenly start to demand a large number of its cycles, degrading the performance perceived by local users. To correct this possible anomaly, the machine could move some of these remote jobs back to their originating nodes or to a new host. In this way control of the local resources could still be maintained. Of course, now the research question becomes how can the load sharing system effectively and efficiently monitor remote jobs. A second new interesting research problem arises due to the recent introduction of multiprocessor workstations to the market. In a network of such workstations, load distribution has to be done at two levels: within the local processors of a workstation and among the workstations.

Finally, we realize that the performance of a computer does not depend solely on its CPU utilization. We have just used this resource to illustrate our ideas. The notions behind Low-mark and High-mark could also be applied to whatever local resource becomes the system bottleneck, such as physical memory or disk space.

Acknowledgements

We would like to thank Peter Potrebic and Phil Goldman who designed and implemented the lsh software.

References

Agrawal1987a.

Agrawal, Rakesh and Ahmed K. Ezzat, "Location Independent Remote Execution in NEST," *Transactions on Software Engineering*, vol. SE-13, no. 8, pp. 905-912, IEEE, August 1987.

Alonso1986b.

Alonso, Rafael, "The Design of Load Balancing Strategies for Distributed Systems," *Proceedings of the U.S. Army Research Office Future Directions in Computer Architecture and Software Workshop*, May 5-7, 1986.

Alonso1986a.

Alonso, Rafael, Phillip Goldman, and Peter Potrebic, "A Load Balancing Implementation for a Local Area Network of Workstations," *1986 IEEE Workstation Technology and Systems Conference Proceedings*, pp. 118 - 124, IEEE Computer Society Press, Washington, DC, March 1986.

Alonso1988a.

Alonso, Rafael and Luis L. Cova, "Resource Sharing in a Distributed Environment," *Proceedings for the 1988 ACM SIGOPS European workshop*, Cambridge, England, September, 1988.

Alonso1988b.

Alonso, Rafael and Luis L. Cova, ''Sharing Jobs Among Independently Owned Processors,'' *Proceeding of the 8th. International Conference on Distributed Computing Systems*, pp. 282-288, Computer Society Press, San Jose, California, June 1988.

Alonso1988c.

Alonso, Rafael and Kriton Kyrimis, ''A Process Migration Implementation for a UNIX System,'' *Proceedings of the Winter 1988 USENIX Conference*, February 1988.

Cova1988a.

Cova, Luis L., ''Load Balancing in Two Types of Computing Environments,'' Tech. Report #CS-TR-165-88, Princeton University Computer Science Department, June 1988.

Eager1986a.

Eager, Derek L., Edward D. Lazowska, and John Zahorjan, ''Adaptive Load Sharing in Homogeneous Distributed Systems,'' *IEEE Transactions on Software Engineering*, vol. SE-12, no. 5, pp. 662 - 675, May 1986.

Ferrari1987a.

Ferrari, Domenico and Songnian Zhou, ''An Empirical Investigation of Load Indices for Load Balancing Applications,'' Tech. Report #UCB/CSD 87/353, University of California at Berkeley Computer Science Division, May 1987.

Hagmann1986a.

Hagmann, Robert, ''Process Server: Sharing Processing Power in a Workstation Environment,'' *Computing Systems.*, pp. 19-23, IEEE Society, Cambridge, May 1986.

Kleinrock1976a.

Kleinrock, Leonard, *Queueing Systems,* Volume 2: Computer Applications, p. 2,13, John Wiley and Sons, 1976.

Leffler1984a.

Leffler, S., W. Joy, and K. McKusick, *4.2 BSD System Manual,* Computer Systems

Research Group, University of California, Berkeley, 1984.

Leland1986a.

Leland, Will E. and Teunis J. Ott, ''Load-balancing Heuristics and Process Behavior,'' *Proceedings of PERFORMANCE '86 and ACM SIGMETRICS 1986*, pp. 54-69, May 1986.

Litzkow1987a.

Litzkow, Michael J., ''Remote Unix: Turning Idle Workstations into Cycle Servers,'' *Proceedings of the 1986 Summer USENIX Technical Conference and Exhibition*, pp. 381-384, Phoenix, Arizona, June 1987.

Metcalfe1976a.

Metcalfe, R. M. and D. R. Boggs, ''Ethernet: Distributed Packet Switching for Local Computer Networks,'' *CACM*, vol. 19,7, pp. 395-404, July 1976.

Mutka1987a.

Mutka, Matt W. and Miron Livny, ''Scheduling Remote Processing Capacity in a Workstation-Processor Bank Network,'' *Proceedings of the 7th. International Conference on Distributed Computing Systems*, pp. 2-9, Computer Society Press, Berlin, West Germany, September 1987.

Mutka1987b.

Mutka, Matt W. and Miron Livny, ''Profiling Workstation's Available Capacity for Remote Execution,'' Tech. Report #697, University of Wisconsin-Madison Computer Science Department, May 1987.

Ni1985a.

Ni, Lionel M. and Kai Hwang, ''Optimal Load Balancing in a Multiple Processor System with Many Job Classes,'' *IEEE Transactions on Software Engineering*, vol. SE-11, no. 5, pp. 491-496, May 1985.

Nichols1987a.

Nichols, David A., ''Using Idle Workstations in a Shared Computing Environment,''

Operating System Review, vol. 21, no. 5, pp. 5-12, ACM Press, November 1987.

Stankovic1984a.

Stankovic, John A., ''Simulations of Three Adaptive, Decentralized Controlled, Job Scheduling Algorithms,'' *Computer Networks*, vol. 8, no. 3, pp. 199-217, North-Holland, June 1984.

Theimer1985a.

Theimer, Marvin M., Keith A. Lantz, and David R. Cheriton, ''Preemptable Remote Execution Facilities for the V-System,'' *ACM*, vol. 1, pp. 2-12, 1985.

Wang1985a.

Wang, Yung-Terng and Robert J. T. Morris, ''A Survey and Comparison of Load Sharing Strategies in Distributed Computer Systems,'' *New World of the Information Society, Proceedings of the 7th International Conference on Computer Communication*, pp. 392 - 393, North-Holland, Amsterdam, Netherlands, 1985.

Williams1983a.

Williams, Elizabeth, ''Assigning Processes to Processors in Distributed Systems,'' *IEEE parallel Proceedings*, pp. 404-406, 1983.

Zhou1988a.

Zhou, Sognian, ''A Trace-Driven Simulation Study of Dynamic Load Balancing,'' *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1327-1341, IEEE Computer Society Press, September 1988.

At each node:

When a local job is invoked:

IF local_load > High-mark **THEN**

BEGIN

executing_node = location_policy();

<request execution at executing_node>;

IF <request accepted> **THEN**

<transfer job to executing_node>;

END

ELSE

<execute job locally>;

When an executing request arrives to a node:

IF local_load < Low-mark **THEN**

BEGIN

<accept request>;

<receive remote job>;

<execute remote job>;

END

ELSE

<reject request>;

Figure 1: The High-Low algorithm

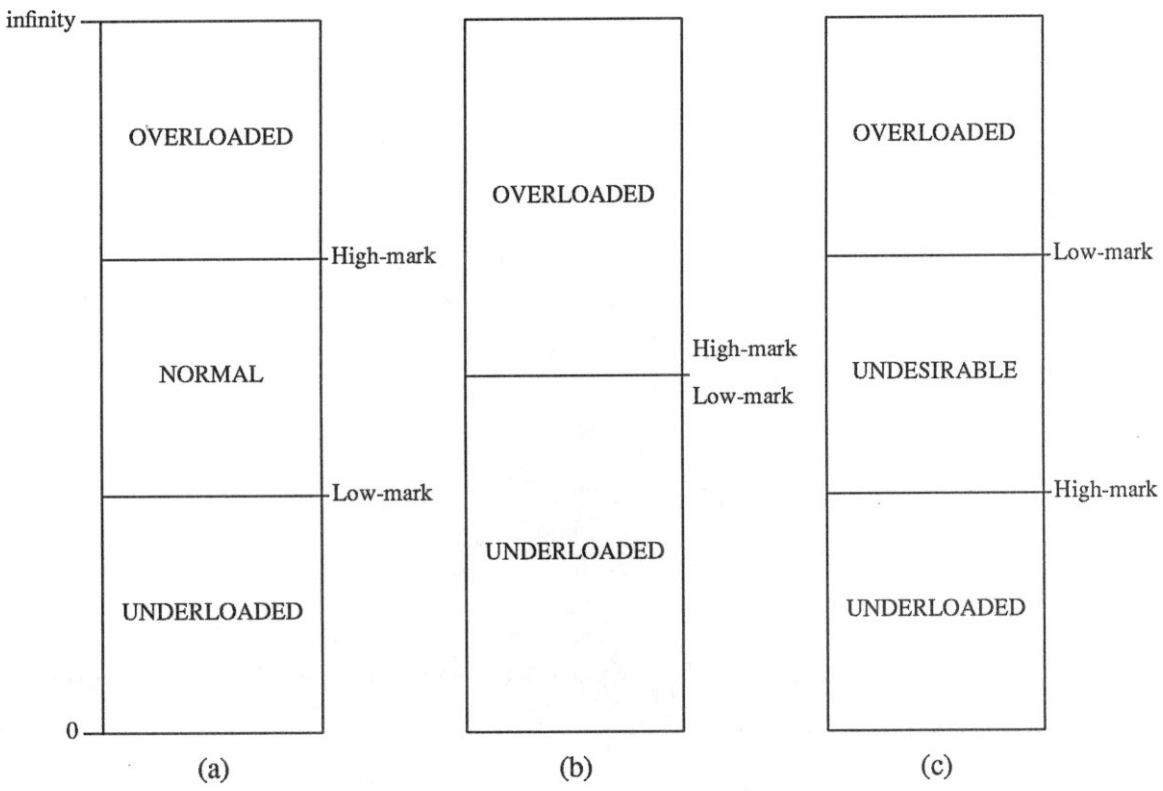

Figure 2: load regions

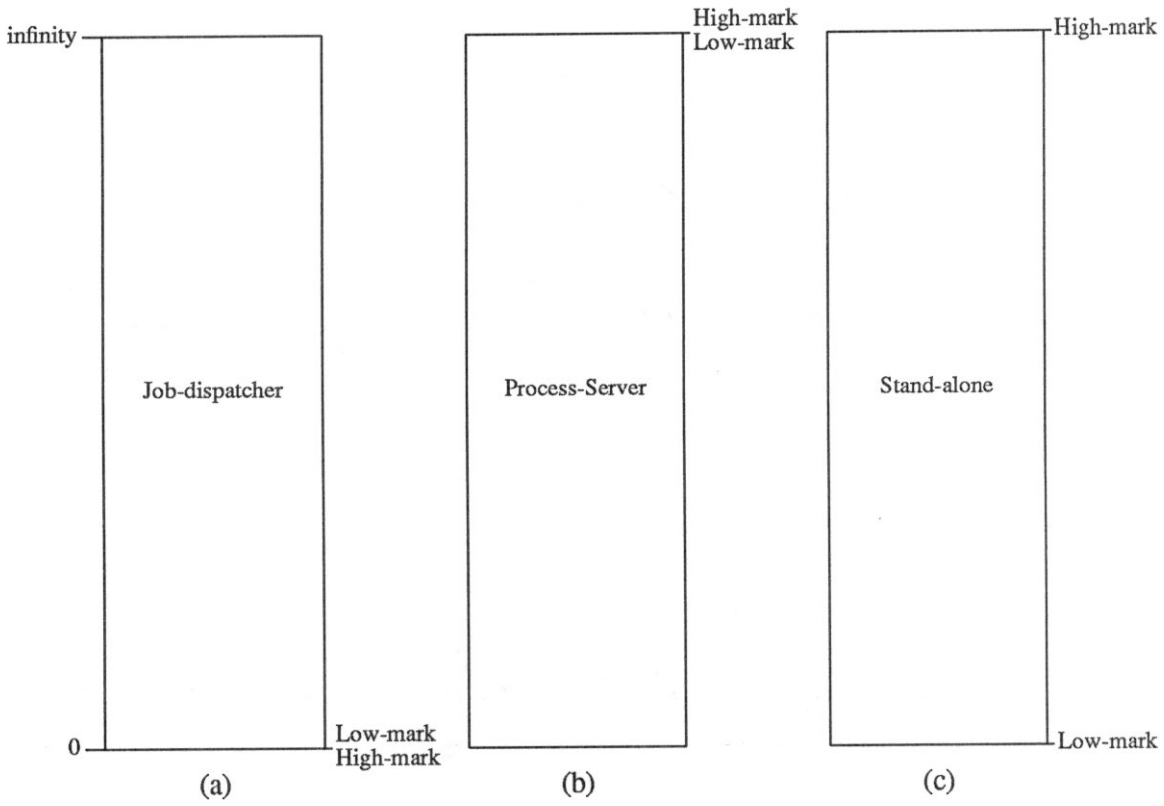

Figure 3: Possible modes of operation

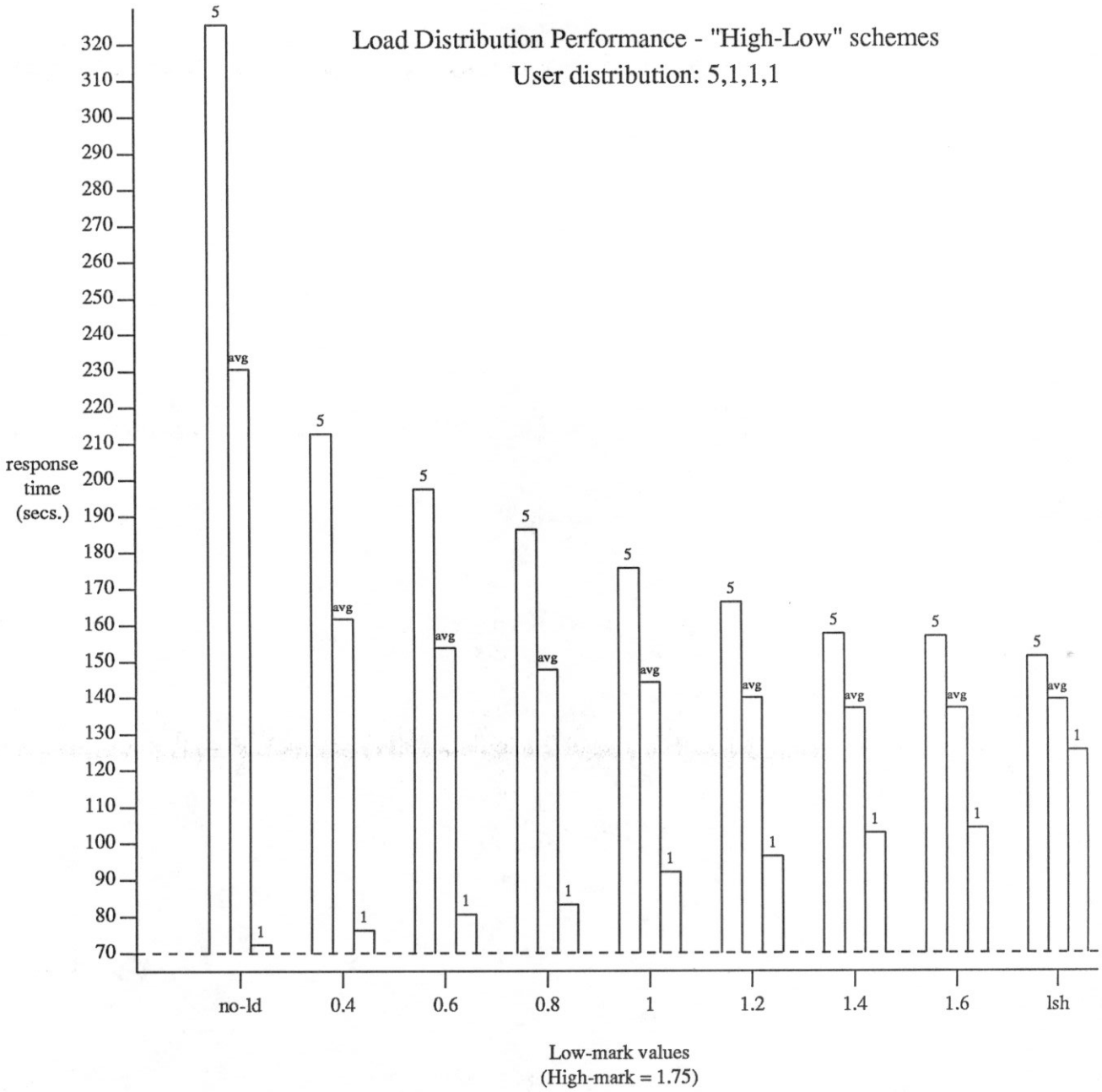

Figure 4: Response time of user jobs using High-Low under a low system load.

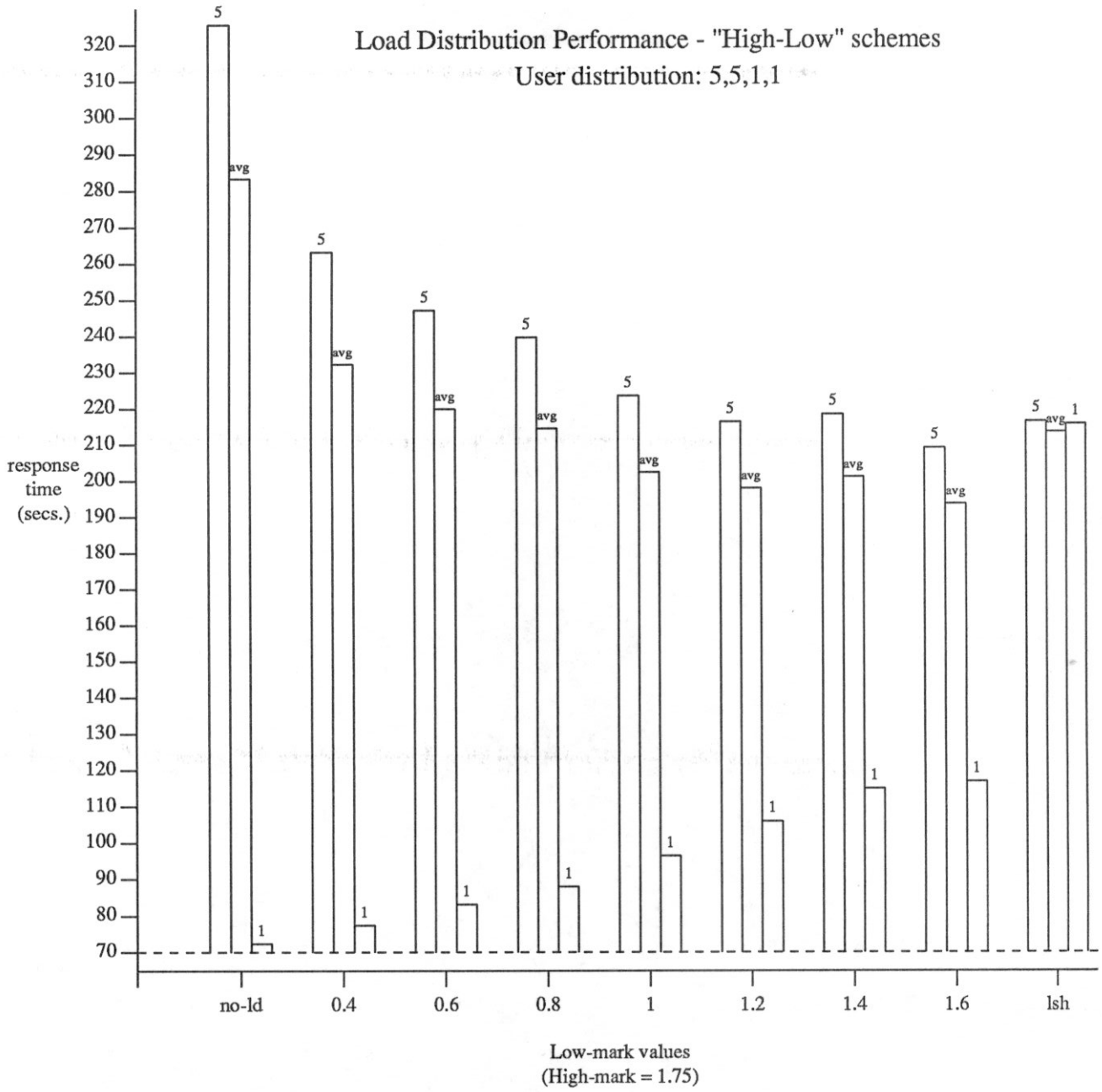

Figure 5: Response time of user jobs using High-Low under a medium system load.

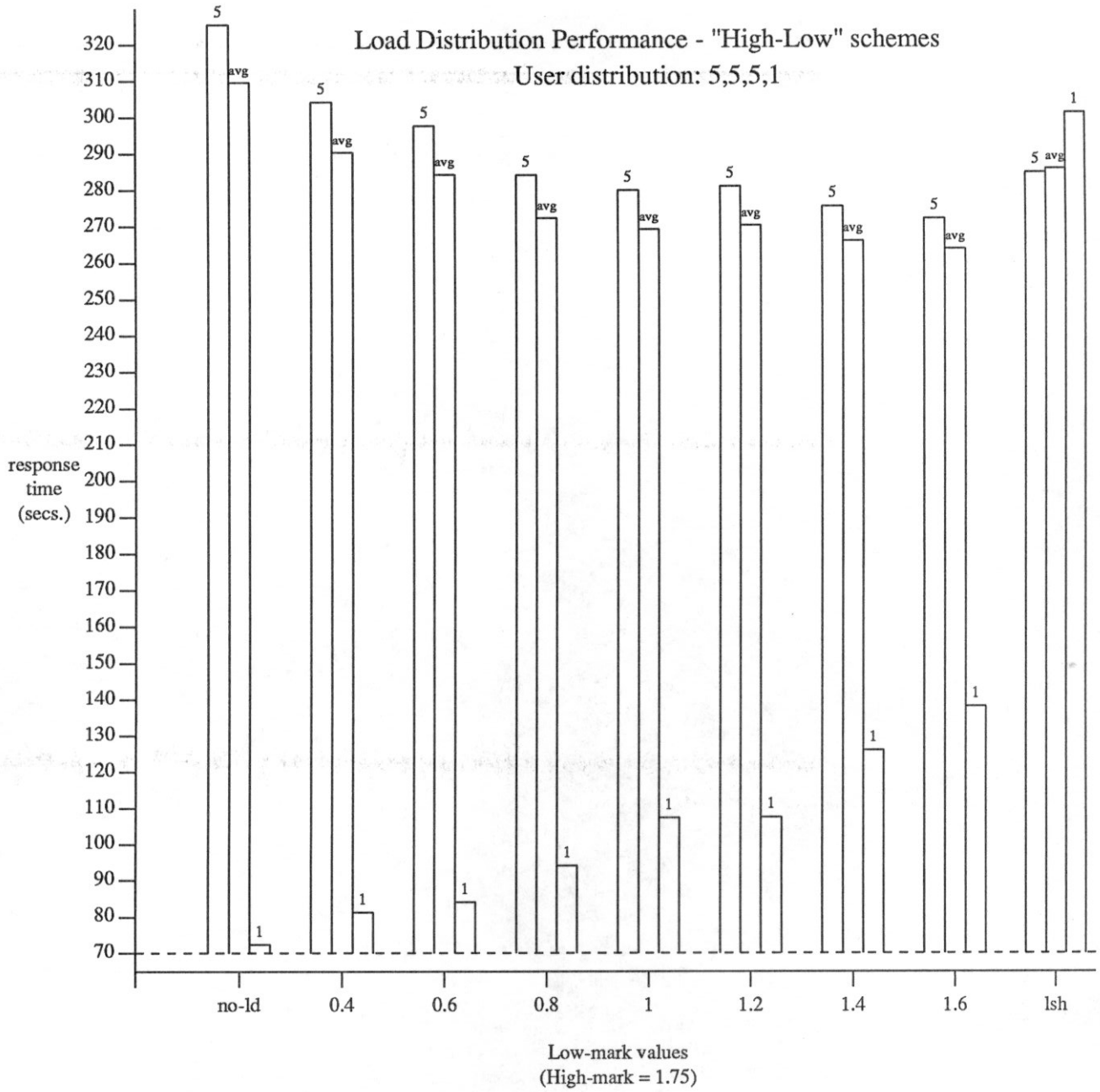

Figure 6: Response time of user jobs using High-Low under a high system load.

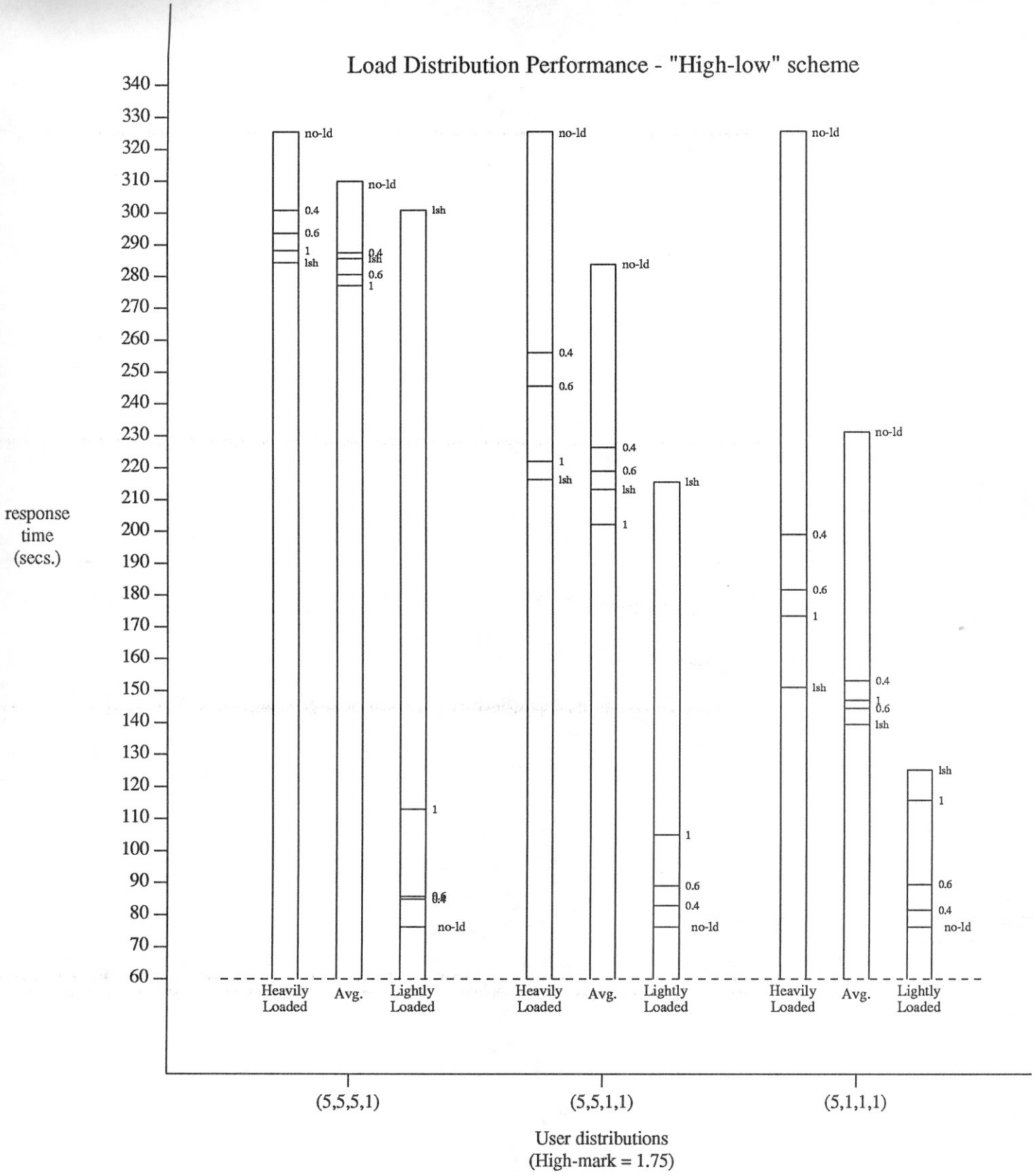

Figure 7: Comparison of different settings for High-Low using the lsh set of policies and under different system loads

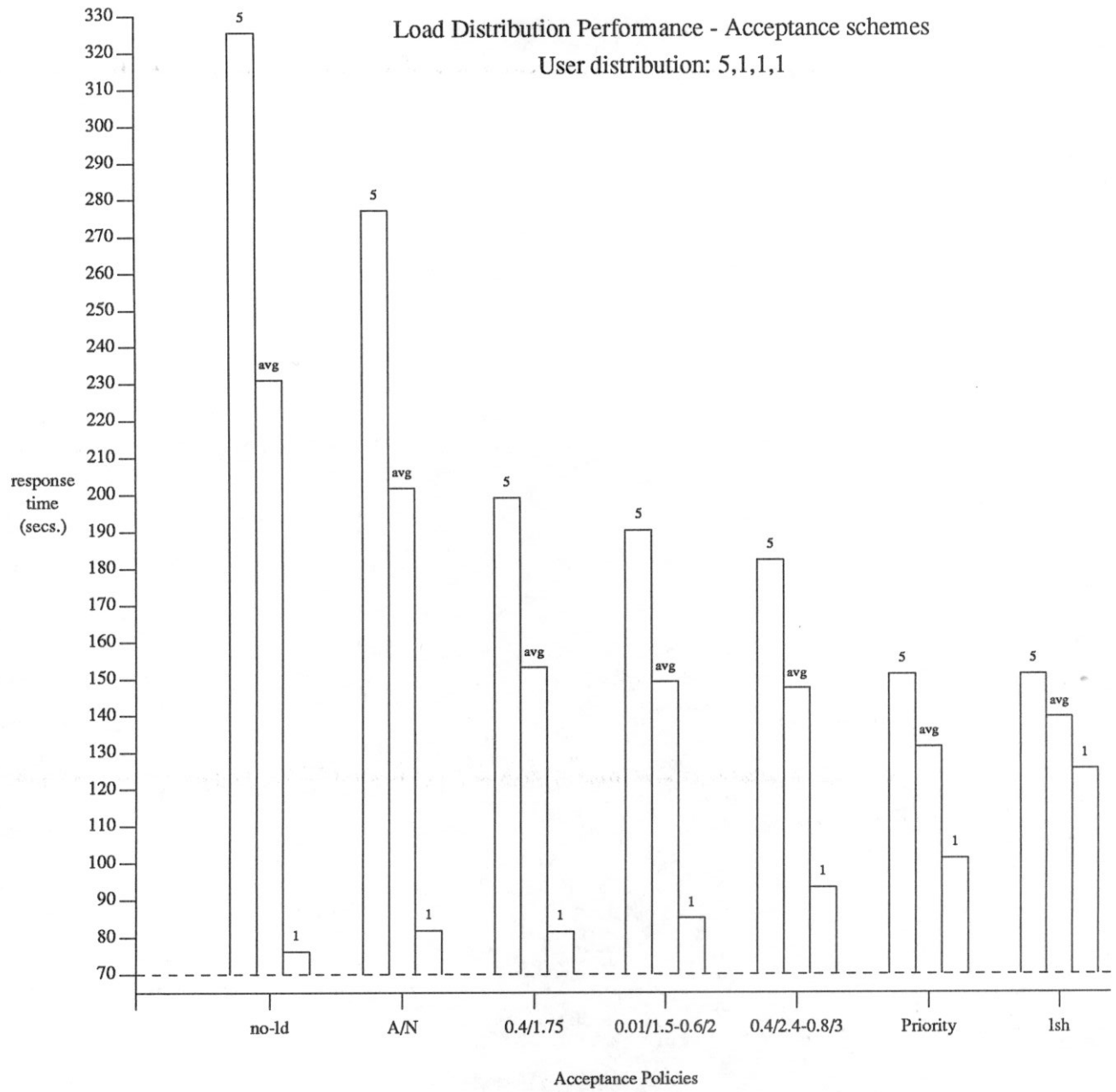

Figure 8: Comparison of different acceptance policies using the lsh set and under a low system load

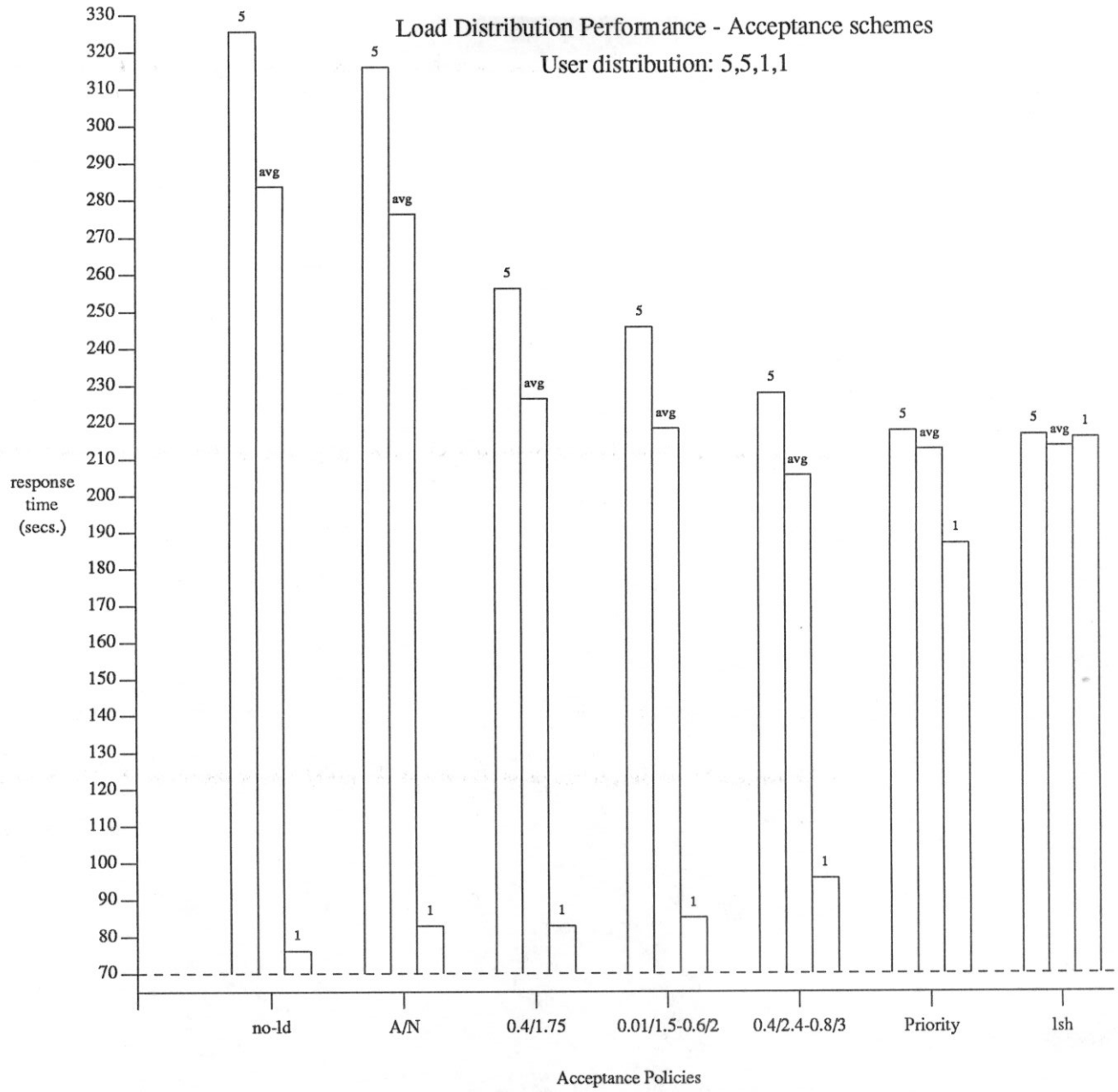

Figure 9: Comparison of different acceptance policies using the lsh set and under a medium system load

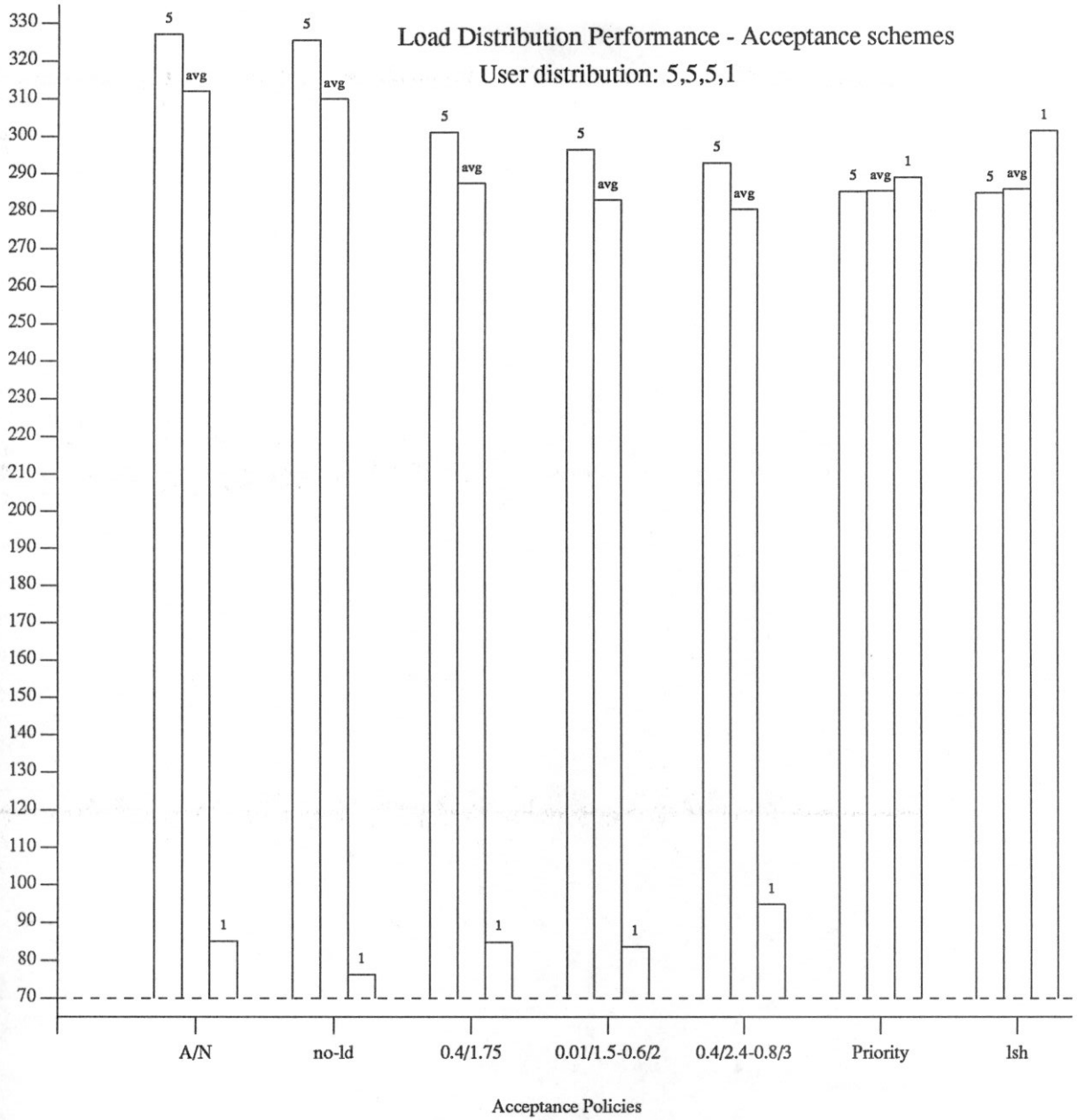

Figure 10: Comparison of different acceptance policies using the lsh set and under a high system load

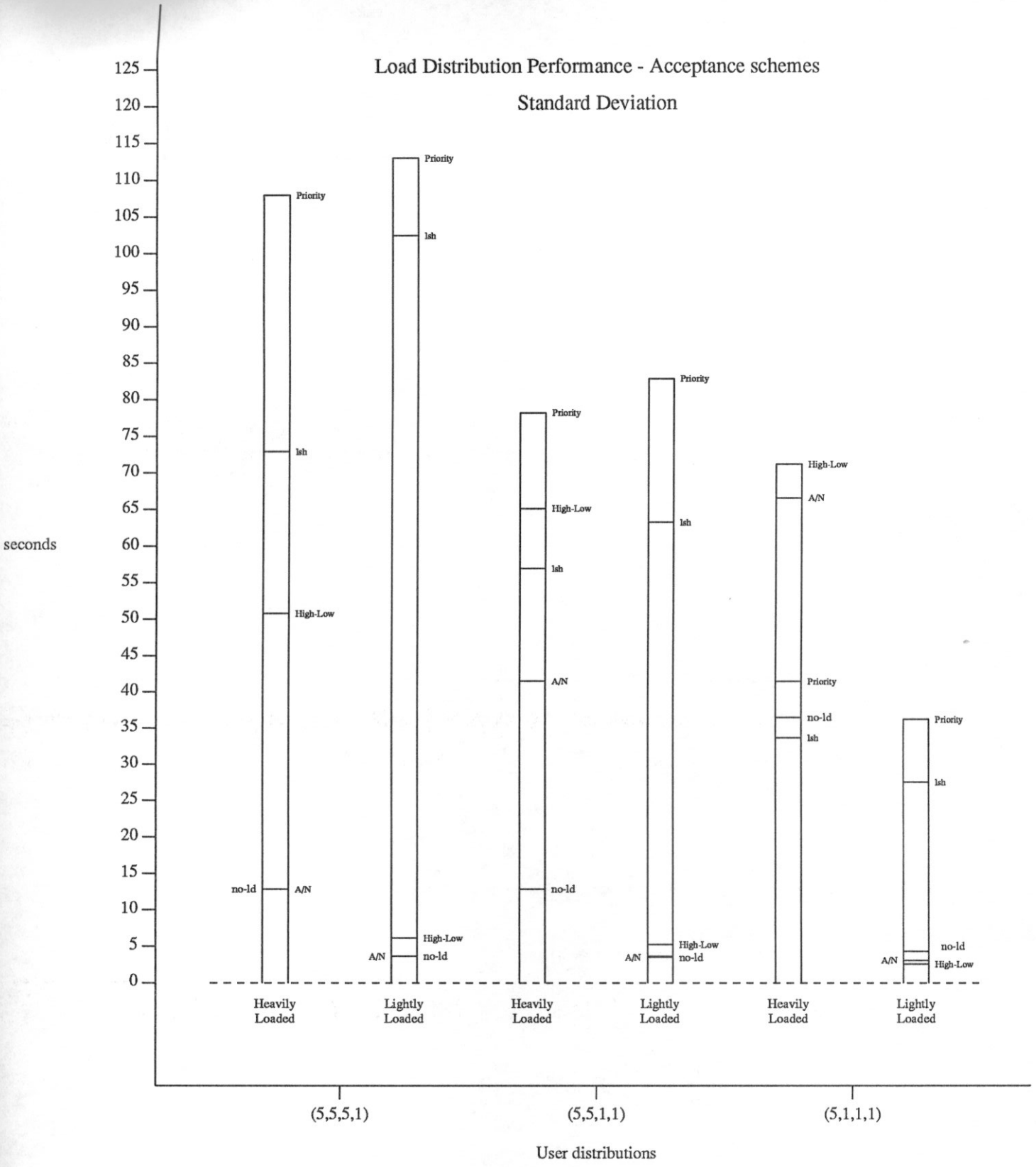

Figure 11: Standard deviation of
different acceptance policies using the lsh set