A LOAD BALANCING IMPLEMENTATION FOR A LOCAL AREA NETWORK OF WORKSTATIONS

Rafael Alonso
Phillip Goldman
Peter Potrebic

CS-TR-018-86

January, 1986

1986 IEEE Workstation Technology and Systems Conference
March 18-20, 1986
Atlantic City, N.J.

A LOAD BALANCING IMPLEMENTATION FOR A LOCAL AREA NETWORK OF WORKSTATIONS

Rafael Alonso, Phillip Goldman, Peter Potrebic

Department of Computer Science
Princeton University
Princeton, N.J. 08544
(609) 452-3869

ABSTRACT

This paper presents the details of a prototype load balancing package and tabulates the results of preliminary performance measurements. Our implementation uses the networking facilities provided in 4.2 BSD UNIX, and is currently running on a local area network of SUN workstations. Although our work is still in progress, initial benchmarks show that our approach is extremely promising.

January 5, 1986

A LOAD BALANCING IMPLEMENTATION FOR A LOCAL AREA NETWORK OF WORKSTATIONS

Rafael Alonso, Philip Goldman, Peter Potrebic

Department of Computer Science
Princeton University
Princeton, N.J. 08544
(609) 452-3869

1. Introduction

In recent years, advances in technology have made possible a distributed computing environment composed of a large number of loosely coupled heterogeneous processors connected by a high bandwidth communication network. This environment is particularly desirable to many users for a variety of reasons which have been widely discussed in the literature (see [LeLann1981] for example). These motivations include giving groups of users the ability to tailor their working environment, gradually increasing the total computational power available, and achieving greater availability, resource sharing, as well as increased performance. Many of the currently existing distributed systems are decentralized, they use a local area network (LAN) [Clark1978] as the communication mechanism, and are composed of a large number of workstations and mainframes. These systems typically support a wide variety of applications. The heterogeneity of the workload, coupled with the decentralization of resource management in the system, can lead to a situation where, most of the time, the system load is quite unbalanced across the nodes of the network. For example, at our local computer center, experience has shown that, while some processors are so heavily loaded as to be unusable, other machines are underutilized. Although often users could then log onto another machine, the resulting control system may be unstable, and naive users cannot be expected to cope effectively with this potential problem. In order to remedy this problem in a general way, load balancing techniques that can be used transparently by the system are required, much in the same way that virtual memory techniques are currently employed on the users' behalf to help them manage their memory space. The understanding of such strategies is the long-term goal of our current research.

A condition of load imbalance is undesirable for many reasons. Apart from the inherent unfairness of that situation (since many jobs are suffering degraded services while others enjoy underutilized resources), an imbalance may cause a decrease in the global system's throughput and an increase in the system's mean response time. Misused resources represent a waste of money and typically result in installations buying more hardware than they really need. It should be made clear that the load imbalance problem exists not only in environments where there is a large number of distributed applications, but even where processes run more or less independently in a single processor, since in the latter case too many processes may be assigned to a given processor. Of course the problem is worse in the former case since a single resource

This research partially supported by New Jersey Governor's Commission Award No. 85-990660-6

bottleneck can potentially affect many distributed applications.

In broad terms, our current research focuses on the following two questions: (1) can the performance of a distributed system be improved by using load balancing techniques?, and if so, (2) how to best implement load balancing? With respect to the first question, it is clear that some degree of improvement could always be achieved by giving extensive load and task information to a global scheduling mechanism, but only if the gains are sizable will this approach be of more than academic interest. Also, it must be shown that, using easily implementable load metrics, and in environments that are not extremely imbalanced, load balancing still makes sense, even when accounting for the overhead of the scheme. Furthermore, it would be desirable to reduce to a minimum the changes required by the introduction of a load balancing scheme. Hence, a better statement of the the first query would be: can load balancing strategies be easily introduced into current distributed systems in such a way that the performance of the system will be substantially improved under reasonable conditions? Our work to date has addressed only this first question.

Answering the second question requires exploring different load balancing strategies, which involves studying the performance of various metric-policy combinations. But it also requires a careful definition of what is meant by the load of a processor (it is clear that both the intrinsic power of a processor and the type of jobs running in it must play a part), as well as a thorough study of the impact of system topology and network bandwidth on the different strategies.

Much of the previous work in load balancing has consisted of formalizations of the problem so that it becomes mathematically tractable (for example, see [Stone1977] and[Chu1980]). We have instead chosen to study the problem by actually implementing a load balancing mechanism and measuring its performance. Although there have been a few load balancing implementations reported (for example, see [Presotto1982] and[Hwang1982]), no attempt was made to study the problem systematically in those cases.

In this paper we describe the nature of our load balancing implementation and give the performance results that indicate that load balancing techniques can be very successfully employed in distributed systems. In the next section we describe both the hardware and software environment of the facility in which our experiments were carried out. Section 3 provides a description of the load balancing software. In Section 4 we report the results of our preliminary tests. In the last section we present our conclusions and remark on the direction of our current research.

2. Experimental Facilities

Our experimental facility is currently composed of 7 Sun workstations of various types: one Sun2/170 with 300 Megabyte disk (which acts as a network file server), four Sun2/120 workstations (each with a 120 Mbyte and a 70 Mbyte disks), and 2 Sun2/50 (which are diskless workstations that page from the file server). All the processors are connected by a stand-alone 10 Mbit/s Ethernet. There were no users on the machines or using the network while our experiments were taking place.

The operating system for the workstations is Sun UNIX 4.2 Release 2.0, a variation of Berkeley UNIX 4.2 BSD [Leffler1983]. We will not describe in detail the networking facilities provided by 4.2 BSD (see[Sechrest1984] for a tutorial overview). Suffice it to say that two communication facilities were used in our work, TCP/IP and UDP. The TPC/IC software enables processes to communicate by using error-free data

streams (the user or application program is guaranteed to receive any message sent to his host), while UDP is a datagram facility (i.e., there is no assurance that UDP messages will arrive at receiving node; we have found that, at high loads, sometimes over 80% of all UDP datagrams are lost).

A few words are in order about Sun's network file system. Sun Release 2.0 supports a network file system called **NFS,** which provides an extension to the standard UNIX tree-like file system structure [Ritchie1978]. (More details about **NFS** are available in [Walsh1985] **NFS** allows each machine to access any file or device in the network with almost complete location transparency. **NFS** accomplishes this with the use of two primitives, *mount()* and *export()*. The former allows machines to actually mount a remote file system as if it were a directory tree; this tree can be linked at any point on the local file system. The latter primitive allows a machine to specify which file systems can be mounted remotely by other machines. This functionality creates a fairly uniform naming environment which simplifies the task of the load balancing software.

3. The Load Balancing Software

Essentially, our prototype implementation consists of a shell (**lsh**) that executes commands in the "least loaded" processor (a shell, in UNIX terms, is an on-line command interpreter); this shell is aided in its task by two ancillary daemons (**lshd** and **rld**) which will be described below.

The load balancing shell is invoked by the call:

lsh command [args]

At this point, from a user's perspective, most commands will proceed to execute as if they were being executed by one of the standard UNIX shells; in particular, the command can be combined with other commands using pipes, have its input and output redirected, and the process may also access environment variables. For almost all applications the load balancing shell is transparent. Of course, there are still some cases in which the shell is not completely transparent; two such cases involve accessing local files (i.e., files which have not been remotely mounted), and communicating with a local process (such as a daemon that is only installed on certain machines).

The interactions between the software involved in the load balancing mechanism are shown in Figure 1. The user first submits his command to the **lsh** shell. The next step in the process is to determine on which computer the job should be run. To do this, **lsh** connects to **rld** (using a virtual circuit) and passes the contents of the command line to it. Based on that information and on data about the load on all the machines in the network, **rld** sends back to **lsh** the name of the machine on which the job is to be run. If the machine chosen is the local one, **lsh** interprets the command in the usual way. However, if a remote host is chosen, **lsh** attempts to connect to the **lshd** daemon on that host. If the connection is successful, **lsh** transfers the command line and other relevant information (such as the user name, current working directory, and the environment variables) to that daemon. If the connection fails then the command is executed locally. At this point, **lsh** has to coordinate with the remote **lshd** in order to to ensure that I/O flows between the command process and the user, and to clean up after the command terminates.

There is a single **lshd** daemon per machine, which forks a process for each request that it receives. The task of the forked process is to ensure that the migrated command sees an environment identical to the one in the original processor.

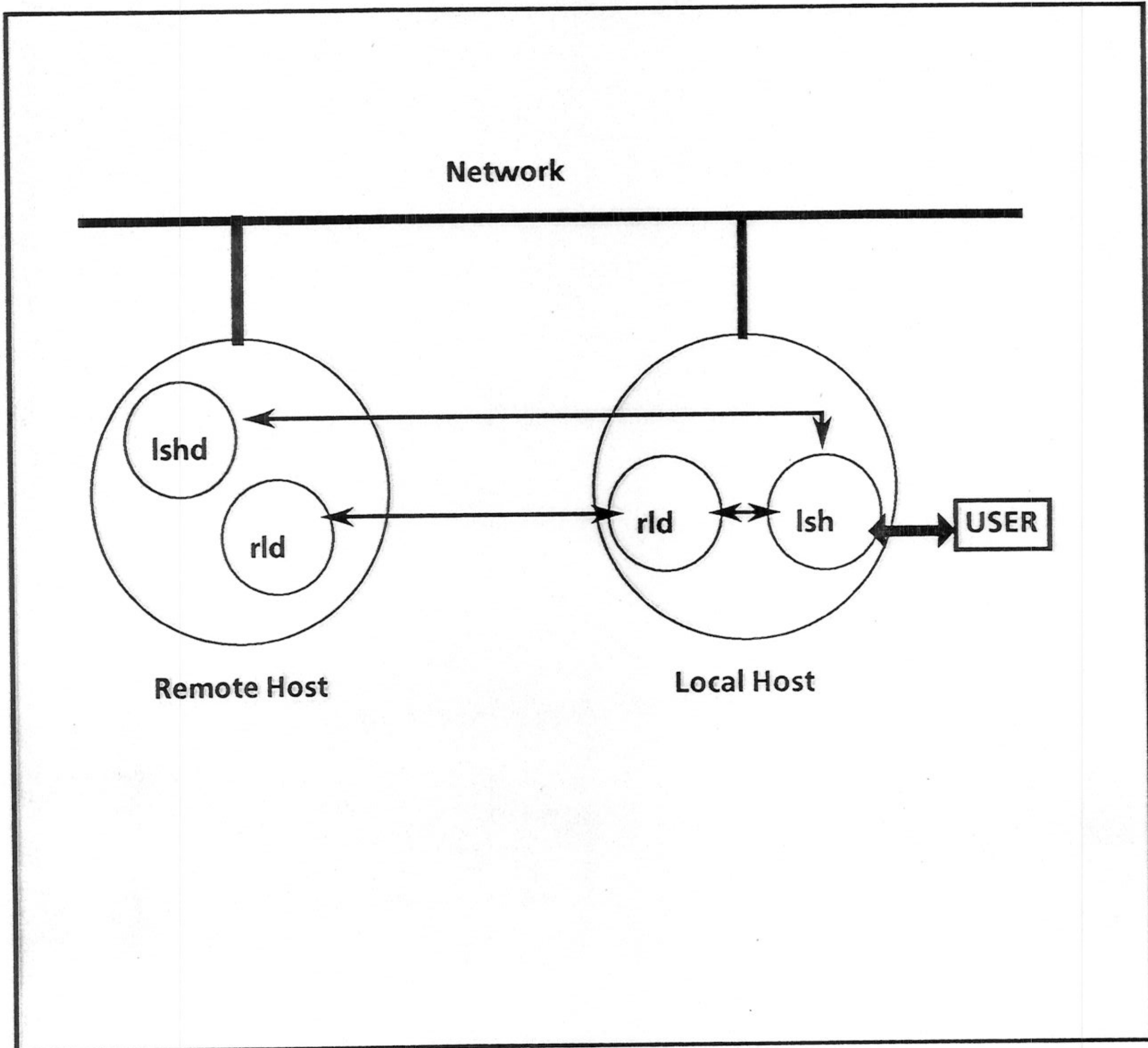

Figure 1. Software Interactions

There is also a single **rld** daemon on each machine in the network. **rld** is responsible for broadcasting the local load information across the network, and maintaining a small database of load data that is has received from the other workstations on the network. (**rld** uses datagrams to distribute local load information across the network.) Broadcasting of the local load data is done at periodic intervals (currently, this interval is 20 seconds long).

The present version of **rld** uses a very simple and naive load metric and decision making algorithm (one of the areas of our current research is the development of more sophisticated definitions of load). **rld** uses as a measure of system load the UNIX "load average", which is defined as the average number of jobs in the run queue over the last 1, 5, and 15 minutes. For our experiments we used the average over the past minute (and we will be referring to this number when speaking of the load average in the rest of the paper). For each remote host, **rld** keeps a time stamp that represents the last time it received load data from that particular machine. If **rld** determines that a particular workstation has not sent its load within the last broadcast interval, it assumes that the workstation is down (until that machine resumes its broadcasts).

4. Performance Results

The first test performed was a measurement of the overhead incurred by running the load balancing system (i.e., the extra load caused by running the two daemons, **lshd** and **rld**). We expected this overhead to be minimal (since the daemons are suspended when waiting for user requests, and the exchange of load information is not very frequent), and our measurements confirmed this. Running the load balancing software increases the load average (defined in Section 3) by 0.02. Low overhead was one of our design goals, since we did not want users to pay a performance penalty when they were not using the load balancing mechanism. This test was performed on a Sun2/120.

There is another kind of overhead, that seen by users of **lsh** when it decides to run their job locally. Clearly, in that case, the users would pay the overhead of the decision-making software of **lsh** without obtaining any benefits. Furthermore, that decision-making overhead increases as the local machine becomes more loaded (the case in which load balancing may seem most appealing to users). In Figure 2 we show a comparison of the response times of two compilations of a C program ("cc middle.c"), one using **lsh** (which, in this case, is choosing to run locally) and the other using the normal system shell. As can be seen, the overhead in choosing **lsh** ranged from less than 1.0 second to a maximum of 11.0 seconds. (Note: in this and the rest of the graphs in this paper, load is defined as the "load average" described in Section 3; furthermore, the maximum load applied in any of the experiments is 4.0, since a higher CPU load renders a Sun2 workstation almost unusable for interactive work. Finally, system load was created by executing a series of CPU intensive jobs in parallel with our tests.) This test was also carried out on a Sun2 120.

In a third test, we measured the time to execute remotely a trivial program using our software. It took 3.0 seconds to execute remotely a command that simply echoed its arguments. This number provides a limit on the type of jobs that are appropriate candidates for migration; clearly, if a command were to require less than 3 seconds to execute locally, it would not be effective to execute it remotely. As a comparison, we measured the time to run the same command using the remote execution software provided by Sun, the **rsh** command (this software lets the user specify a remote site and execute a command there). We found that it took 5.6 seconds to accomplish the task using that software, even though our mechanism had the extra overhead involved in choosing an execution site. This test was conducted using 2 Sun2/120's.

The rest of the experiments described in this Section were devised to test **lsh** in a "canonical" two machine configuration. That is, there was a "local" processor where the user entered his commands, and a "remote" host (whose load was always lower than the local one). Of course, in a real situation, there would be many "remote" hosts, but as

far as the performance of **lsh** goes, there is only one significant "remote" host (the one that **lsh** chooses as the least loaded one).

Figure 3 shows the response time for "cc middle.c" again, this time executing it both in the normal manner (i.e., locally) and with **lsh** , for different combinations of local and remote loads. Both the local and remote hosts were Sun2/50's (i.e., the diskless workstations that paged from the file server). The line labeled "local cc middle.c" marks the response time of the compile command, executing locally, as the local load increases from 0 to 4. The family of curves labeled "lsh (remote = *i*)" reveal the response time of the same command, but now running under **lsh**, for a variety of remote loads. There are several items to notice in this graph. First, it is clear that, if there is a remote machine that is idle, it is faster to use **lsh** than to execute locally (except when local load is almost zero). Also, if the local load is less than or equal to the load of the least loaded remote site, **lsh** performs only slightly worse than the normal shell (in this case **lsh** is choosing to run locally, but still has to incur the decision-making overhead). And, for any value of local load, if there is a remote machine with a load slightly less than the local load, the user obtains a faster response time by using **lsh**. The actual amount of improvement in response time can be sizable, even when there is a relatively small difference in loads. For example, if the local load is 3 and the remote load is 1, a user could improve the response time of his compile by almost 40 percent.

We repeated the above experiment, this time using 2 Sun2/120's. In this case, the code to be compiled resided in the disk of the local machine, and thus, the code had to be migrated to the remote processor before compilation. As Figure 4 shows, the graph is very similar to that of Figure 3, but there is a small increase in the overhead of using **lsh** when compared to using the standard shell.

In Figure 5 we show the result of a another experiment, this time with a smaller job ("small.c") to be compiled. Two Sun2/50's were again used here. Since the execution time of this command is of the same order of magnitude as the overhead of using **lsh**, the potential gains of using the load balancing mechanism are only realized when the difference in load averages is large; for example, even when the local load is 3 and there is a remote machine with a load of 1, the user can obtain faster response time by running the normal shell.

Figure 6 shows the information from the graph in Figure 3 in a different form. The x-axis of the graph is labeled with the value of the load in the local processor, and the remote load appears on the y-axis. The graph shows the region in the load space where the use of **lsh** is effective (i.e., where using **lsh** leads to an improvement in performance). In Figure 7 we do the same with the information of Figure 5. A comparison of these two graphs shows that, although in both cases there are large regions of the load space for which the use of **lsh** is indicated, the command "cc medium.c" is better suited for load balancing than the other one. The line marked "optimal" in both graphs represents the ideal performance of a load balancing mechanism with zero overhead (for both decision-making and moving data across the network); with such a system, for these two experiments, running on a remote machine with the same load as the local processor should be just as fast as running locally. The overhead of our system can be appreciated as the separation from the optimal line of our results.

5. Conclusions

In the previous Section, we presented the results of some of our performance tests, which show the possible benefits of using load balancing strategies in local area networks. The performance improvements that can be obtained are sizable, and achievable even under conditions of mild load imbalance. We have studied a variety of UNIX commands, in experiments similar to those in Section 4, and the results have been equally satisfying.

It is clear that there are many other issues to be explored in the context of load balancing mechanisms. In particular, we are currently trying to develop less naive load metrics, ones that will also include the notion of disk utilization. We are also exploring the performance of a variety of load balancing schemes, including ones that take into account the type of job being migrated. In this study, we did not focus on network load, but, as distributed applications become more common, and as more users work on diskless workstations that constantly create network traffic, it does not seems improbable that the network might become a potential bottleneck. We have tentative solutions for some of these problems, but much work remains to be done.

References

LeLann1981.
LeLann, Gerald, "Motivations, Objectives and Characterization of Distributed Systems," in *Distributed Systems — Architecture and Implementation: An Advanced Course, Lecture Notes in Computer Science*, ed. B. W. Lampson, M. Paul, and H. J. Siegert, vol. 105, pp. 1-9, Springer-Verlag, 1981.

Clark1978.
Clark, D. D., Pogran, K. T., and Reed, D. P., "An Introduction to Local Area Networks," *Proceedings of the IEEE*, vol. 66, no. 11, pp. 1497-1517, November 1978.

Stone1977.
Stone, H. S., "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 1, pp. 83-93, January 1977.

Chu1980.
Chu, W. W., Holloway, L. J., Lan, M., and Efe, K., "Task Allocation in Distributed Data Processing," *IEEE Computer*, November 1980.

Presotto1982.
Presotto, D., "Dsh command," 4.2 BSD manual page, U.C Berkeley, 1982.

Hwang1982.
Hwang, K., Croft, W. J., Goble, G. H., Wah, B. W., Briggs, F. A., Simmons, W. R., and Coates, C. L., "Unix Networking and Load Balancing on Multi-Minicomputers for Distr. Proc.," *IEEE Computer*, April 1982.

Leffler1983.
Leffler, S., Joy, W., and McKusick, K., *UNIX Programmers's Manual - 4.2 Berkeley Software Distribution,* U.C Berkeley, August 1983.

Sechrest1984.
Sechrest, Stuart, "Tutorial Examples of Interprocess Communication in Berkeley UNIX 4.2BSD," Report No. UCB/CSD 84/191, U.C. Berkeley, June 1984.

Ritchie1978.
Ritchie, D. and Thompson, K., "UNIX Time-Sharing System," *Bell System Technical Journal,* vol. 57, no. 6, pp. 1905-1929, 1978.

Walsh1985.
Walsh, Dan, Lyon, Bob, and Sager, Gary, "Overview of the Sun Network File System," *Proceedings USENIX Winter Conference 1985,* January 1985.

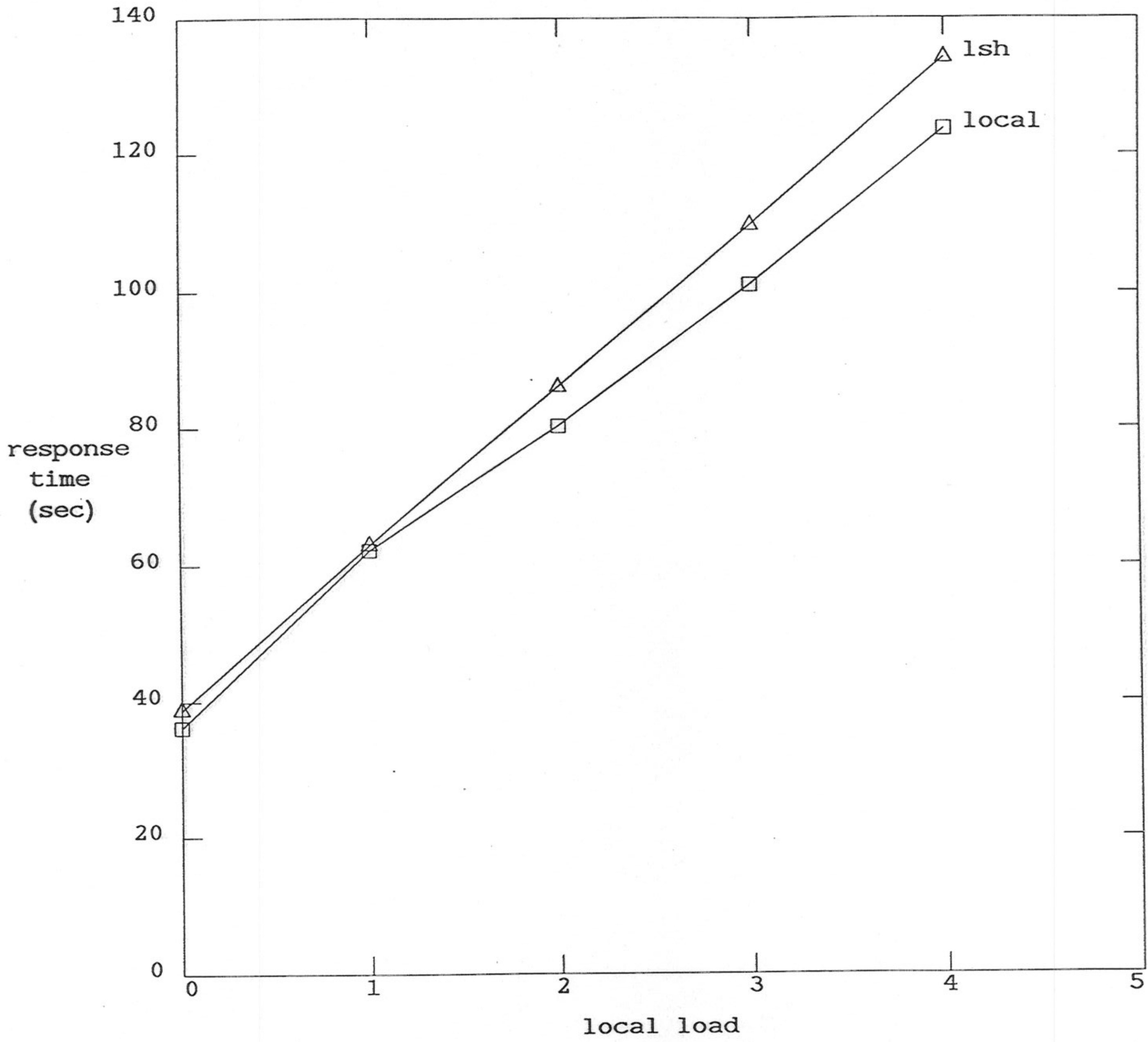

Figure 2. Lsh Overhead - compile of middle.c

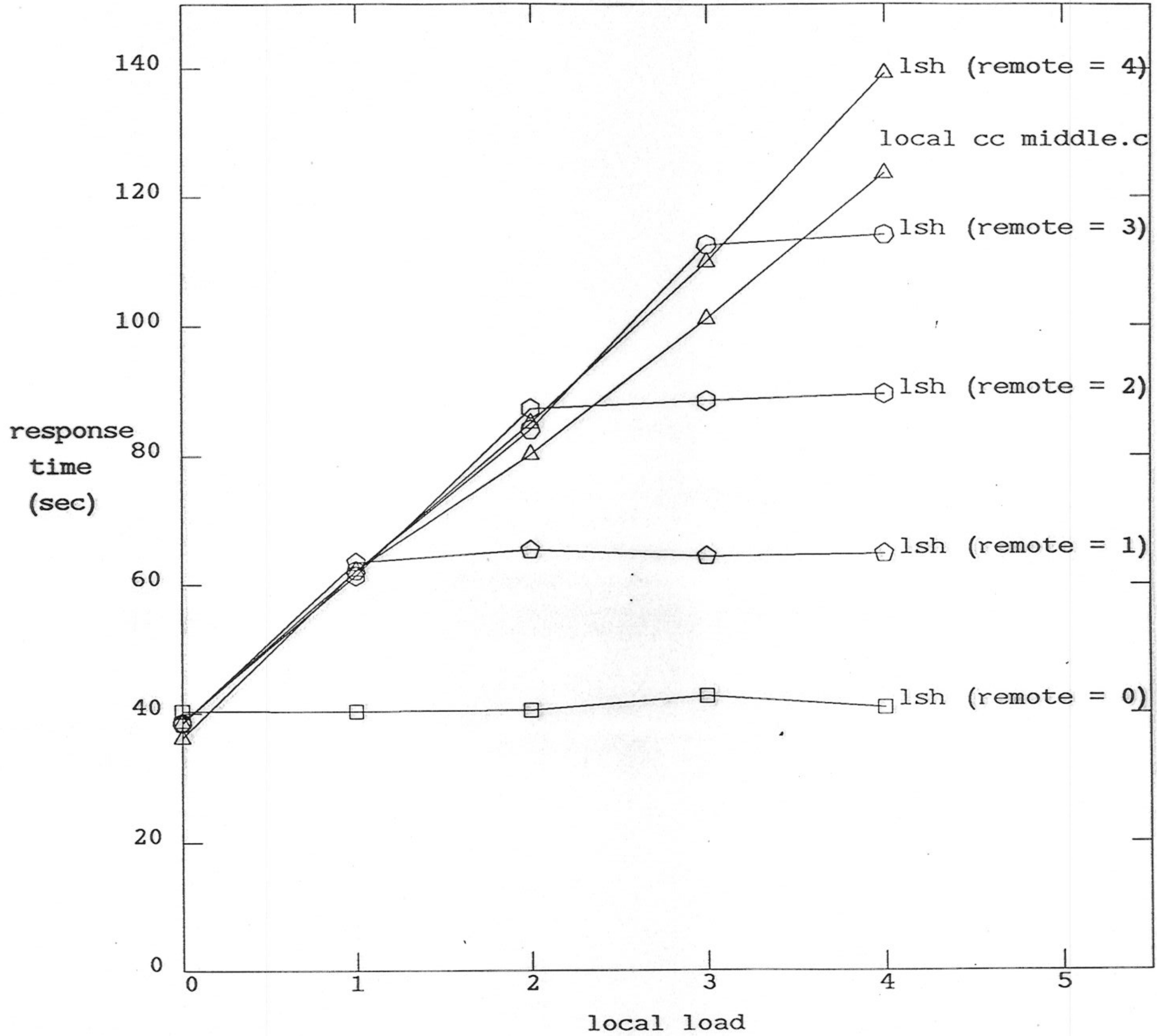

Figure 3. Lsh Performance: compile of middle.c, diskless workstations

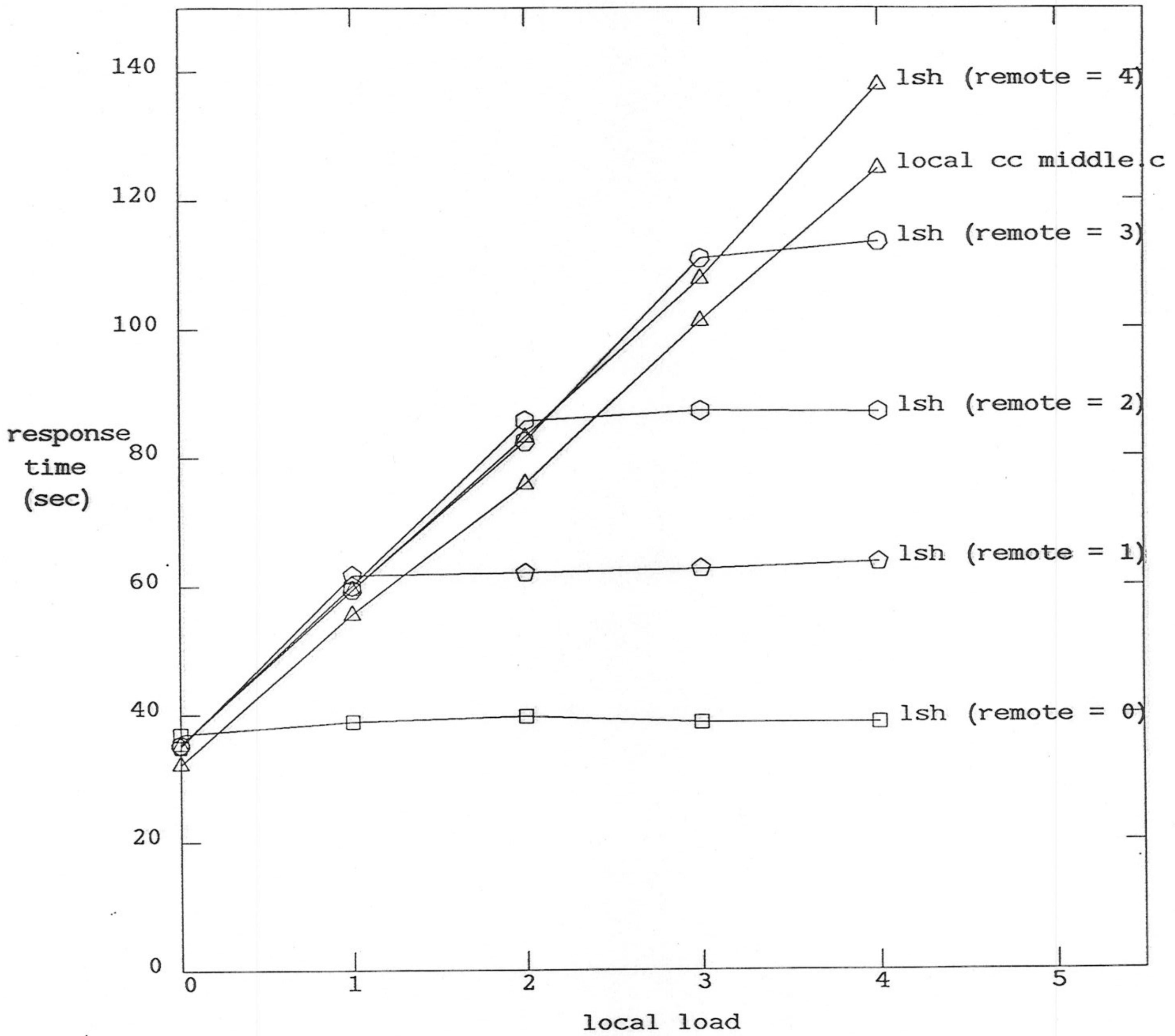

Figure 4. Lsh Performance: compile of middle.c, local disk

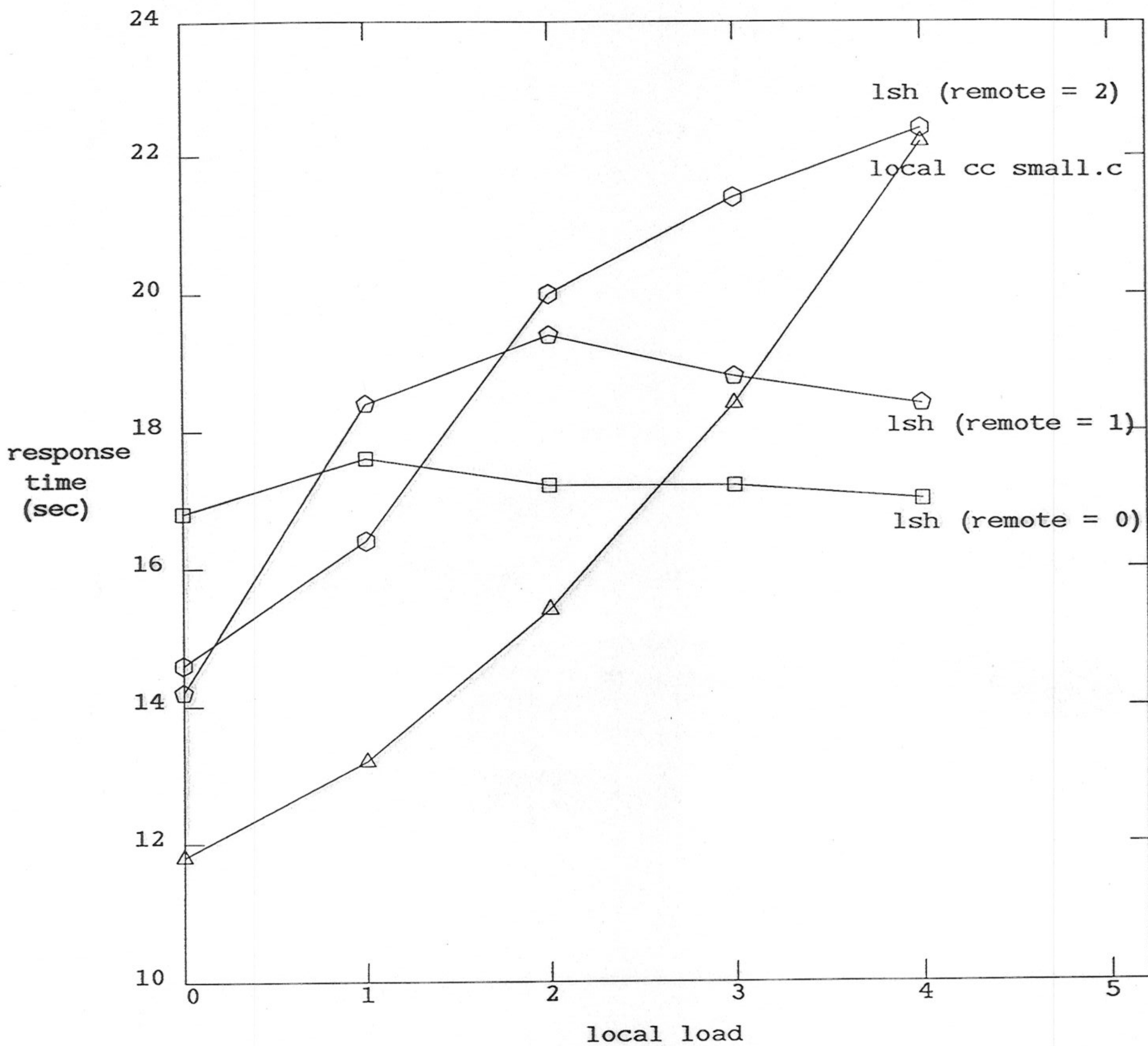

Figure 5. Lsh Performance: compile of small.c, diskless workstations

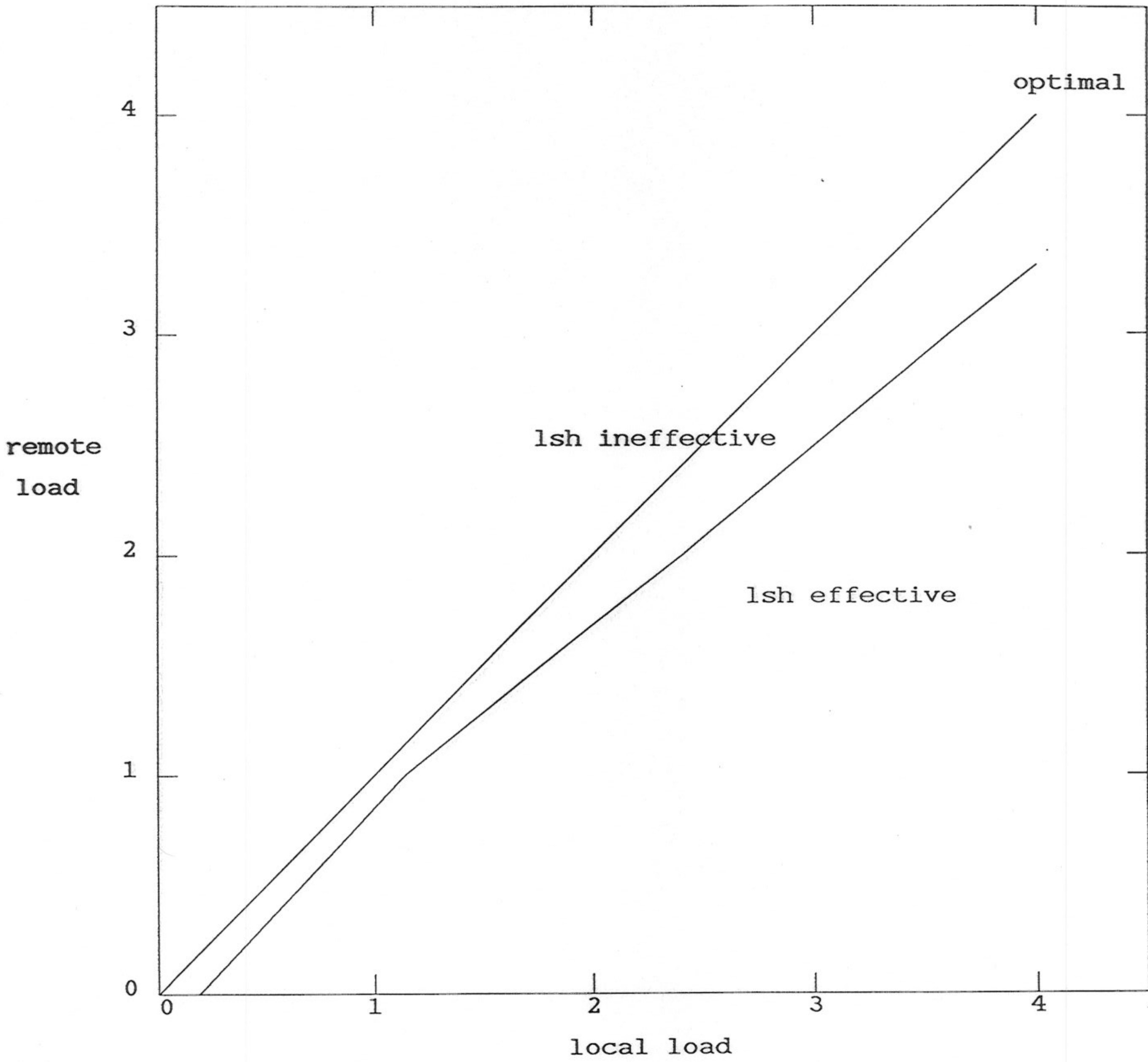

Figure 6. Lsh Effectiveness: compile of middle.c, diskless workstations

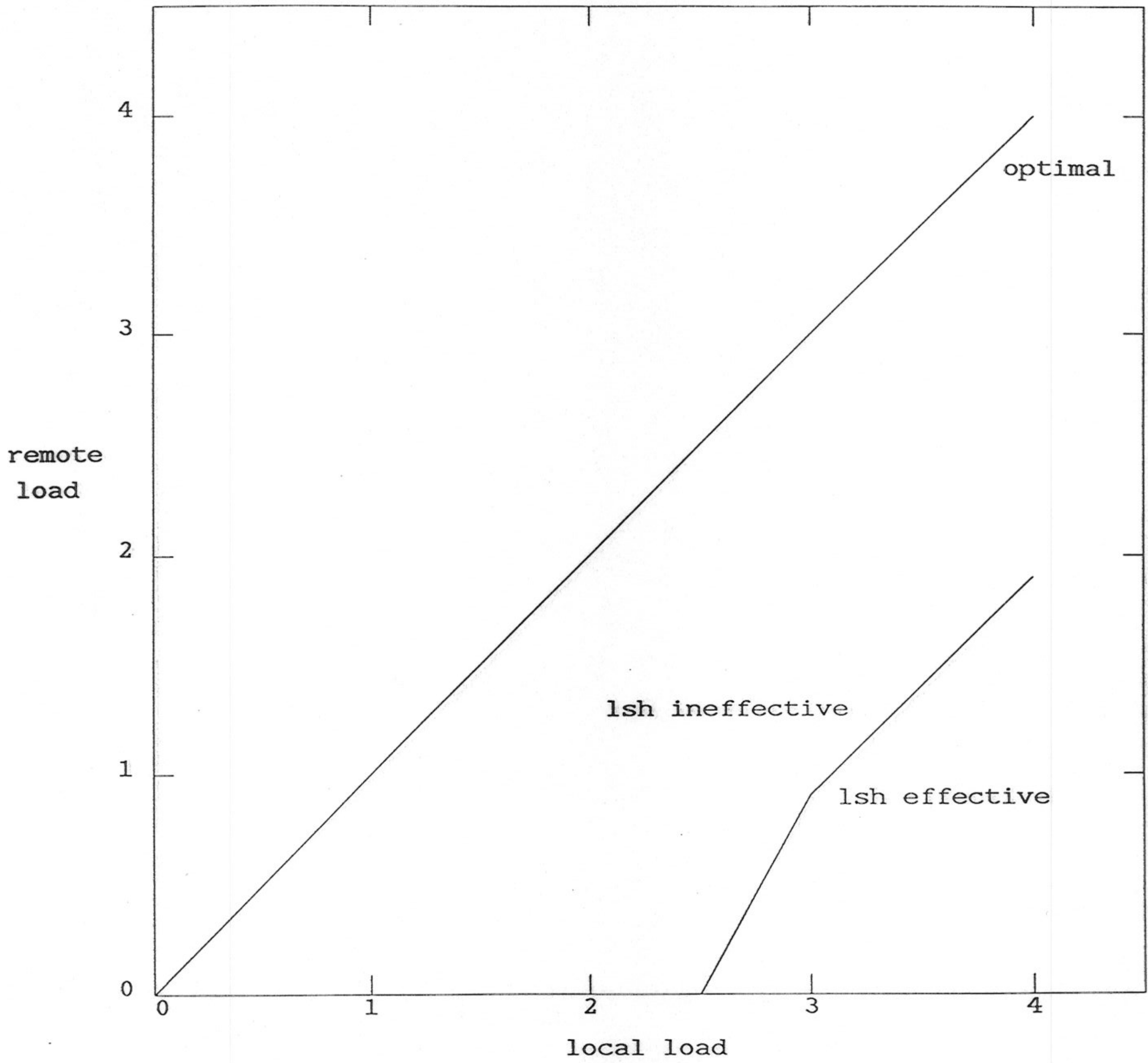

Figure 7. Lsh Effectiveness: compile of small.c, diskless workstations