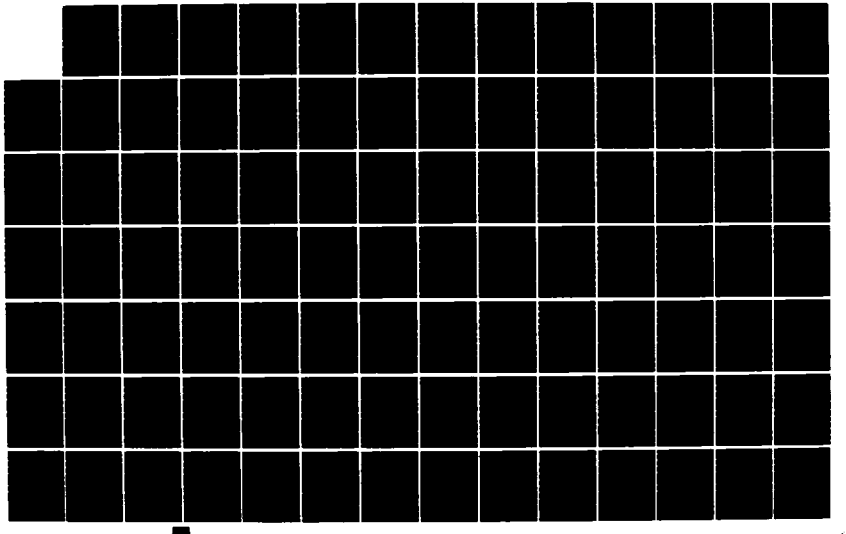
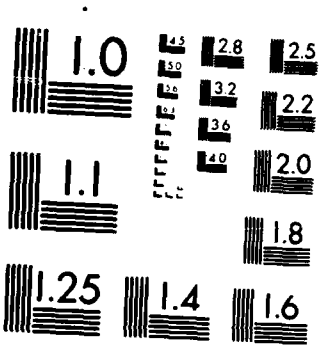
AD-A173 473 QUERY OPTIMIZATION IN DISTRIBUTED DATABASES THROUGH LOAD BALANCING(U) CALIFORNIA UNIV BERKELEY COMPUTER SCIENCE DIV R ALONSO 06 AUG 86 N00039-84-C-0089 1/1

UNCLASSIFIED F/G 9/2 NL





Productivity Engineering in the UNIX† Environment

(9)

Query Optimization in Distributed Databases through Load Balancing

Technical Report

S. L. Graham
Principal Investigator

(415) 642-2059

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government."

Contract No. N00039-84-C-0089

August 7, 1984 - August 6, 1986

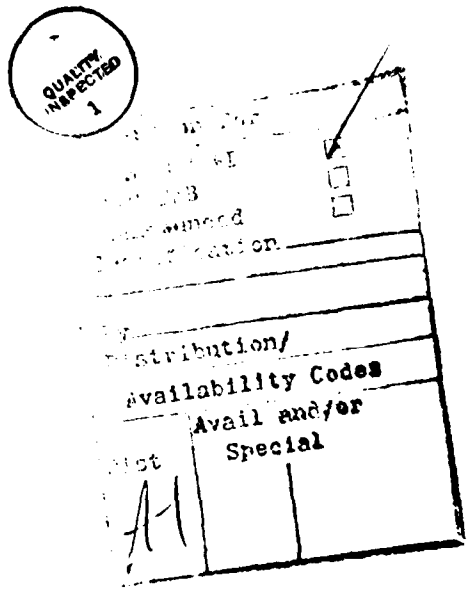
Arpa Order No. 4871

†UNIX is a trademark of AT&T Bell Laboratories

Query Optimization in Distributed Databases Through Load Balancing

by

Rafael Alonso



Query Optimization in Distributed Databases Through Load Balancing

Rafael Alonso

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

As technology advances, computing environments composed of large numbers of workstations and mainframes connected by a high bandwidth local area network become attractive. These systems typically support a large number of application classes, and the heterogeneity of their workload, coupled with the decentralization of the systems, can lead to load imbalances across the network. This work attempts to study the benefits of load balancing strategies in the context of a particular application, distributed database systems. It was felt that, by focusing on a specific area, the problem would become more tractable. The choice of database management systems can be justified not only by their intrinsic importance, but also by the adaptability of load balancing strategies to query optimization algorithms.

In order to determine whether load balancing strategies can indeed be adapted to current optimizers with a moderate amount of effort and to see whether the resulting performance benefits are sizable, both benchmarking and simulation experiments were carried out. The approach taken was first to construct a simple model in order to gain some insight into the problem. This was followed by some benchmarking experiments on a running system, the R* distributed database at the IBM San Jose Research Laboratory. Finally, a model of distributed INGRES was constructed and validated by measurements of INGRES queries and of U.C. Berkeley's TCP/IP implementation. It was hoped that, by utilizing two different techniques, simulation and measurement, and by examining two very different distributed database systems, R* and distributed INGRES, the results of this thesis would be of both greater reliability and wider applicability.

The conclusions of this study are that query optimizers are relatively easy to modify in order to incorporate load balancing strategies, and that the increase in the running time of the algorithms is negligible. Furthermore, load balancing results in a

sizable performance improvement even in environments with a moderate load imbalance. It should be pointed out that the results of this work are important not only from the viewpoint of load balancing studies, but also provide useful insights into the construction of distributed database systems.

ACKNOWLEDGEMENTS

I would like to thank my thesis committee for their guidance in completing this work. Lucien LeCam made many valuable comments concerning the statistical aspects of my thesis. Mike Stonebraker had many insights on the truly important aspects of the problem, and was invaluable in defining the scope of my thesis in the early stages of the work. Most of all, I would like to thank my thesis advisor, Domenico Ferrari, who led me through numerous rewrites, and always made time for me even when he was overloaded with pressing matters.

The working environment at Berkeley was extremely supportive for carrying out research. I would like to thank the members of the Computer Science department for having taught me so much in so many ways. I would specially like to thank the members of the PROGRES and CSR groups, who sat through many trial talks.

A large part of the work involved in my thesis was carried out at the IBM Research Laboratory in San Jose. The members of the R* group deserve special thanks for allowing me to use their facilities for experimentation. In particular I would like to acknowledge the help of Robert Yost, Bruce Lindsay and Guy Lohman.

This work was supported by an IBM Fellowship and by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4871 , monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089 . The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

I would like to thank my parents, Maria Luisa and Matias, for all they have taught me during the years. Most of all, I would like to thank my wife, Marilin, who endured the many hardships that this thesis caused her without complaining, and was always there to encourage me when I needed it most. To her, I dedicate this work.

TABLE OF CONTENTS

CHAPTER 1

Introduction

In recent years, advances in technology have made possible distributed computing environments composed of a large number of loosely coupled heterogeneous processors connected by a high bandwidth communication network. These environments are 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, use a local area network (LAN) [Clark1978] as the communication mechanism, and are composed of a large number (but usually less than 100) of workstations and mainframes. These systems typically support a wide variety of applications. Such 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 U.C. Berkeley, 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 these potential instabilities. 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 manage their memory space automatically. It is the study of automatic load balancing strategies that motivated this research.

In order to make the problem tractable, this work focuses on one particular application, distributed database management systems (DDBMS's). The problem area that this thesis addresses is that of integrating load balancing strategies with the algorithms currently used by DDBMS's so as to improve their performance. Not only are DDBMS's an extremely important application but, as will be shown later, the algorithms employed by query optimizers work particularly well in conjunction with load balancing schemes. Furthermore, there is somewhat more user experience with DDBMS's (although not a large amount) than with other types of distributed systems. It should also be noted that the work in this dissertation cannot only be justified from a load balancing point of view, but also for the insights it brings into the development of DDBMS query optimizers.

The rest of this chapter provides some background information. First, some of the attributes of load leveling are summarized, and its benefits detailed. This is followed by an introduction to the relevant features of DDBMS's and their query optimizers. Next, there is a section providing an extended statement of the problem and commenting on the adaptability of load balancing strategies to the query optimization problem. The subsequent section explains the experimental approach followed. The chapter concludes by presenting an outline of the dissertation.

1.1. Load Balancing

A load balancing strategy is composed of two parts, a load metric and a load policy. The former is an estimator of the level of load of a given processor, and is typically a function of one or more system variables. For example, in Berkeley UNIX[1] 4.2 BSD [Leffler1983a, Ritchie and Thompson 78], the load metric used is the load average (given by both the **la** and the **w** commands), which is loosely defined as the number of jobs in the run queue. The values of the load metric are used by the policy in order to make load balancing decisions. An example of a load policy would be to send incoming jobs to the processor with the smallest load (as defined by the appropriate metric) except for word processing jobs (which always go to a batch machine) and for line-printer jobs (which are always executed locally). There are many possible criteria by which to judge a load balancing strategy, for example, the ease and cost of implementation, and the stability of the algorithms. The purpose of this thesis is not to examine in detail the problem of choosing the load metric and the load policy, but rather to determine the suitability of load balancing strategies for DDBMS's.

A condition of load imbalance is undesirable for many reasons. Apart from the inherent unfairness of that situation (since many jobs are suffering degradation of service while others enjoy the use of 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 bottleneck can potentially affect many distributed applications; it is also a much more difficult problem to solve.

1.2. Distributed Databases and Query Optimization

In the past few years much research has been done in the area of distributed database management systems (DDBMS's): see [Rothnie1980, Williams1982,

[1] UNIX is a trademark of Bell Laboratories

Stonebraker1977] for some examples. DDBMS's are attractive for a variety of reasons: they allow greater security by letting groups within an organization retain control over access to their data; sharing of data and other resources is simpler than in a collection of single site databases; there is the possibility of physical parallelism in the processing of queries; data may be placed closer to where it is used, thereby enhancing performance. While the usefulness of DDBMS's is clear, many of their implementation issues have not been fully resolved, partly because of their complexity, but also because there is limited real experience with their use.

Two of the main decisions to be made in the design of a DDBMS are: how should the data be partitioned among the nodes in the system; and, secondly, how should query processing take place. The answers to both questions are of critical importance to the performance of the DDBMS. The first topic deals with the issues of data replication, availability, access patterns, and consistency, among others. In the present study this area will not be considered, in the sense that data allocation strategies will not be investigated, but of course *how* the data is partitioned will affect our work in the second area. It is the second issue which concerns us, namely, given that the data is distributed in some "reasonable" form, how should the DDBMS handle a user inquiry involving data that is at perhaps widely separate locations so as to satisfy the inquiry with reasonable performance. It should be noted that the physical location of the data is, in most systems, decided upon at database generation time, while query processing is done on a query by query basis throughout the lifetime of the system.

To better understand the optimization problem, we will focus our attention for a moment on the role of a DDBMS query optimizer (we will also refer to it as a query planner). The query optimizer in a DDBMS is presented with a complex query that may deal with multi-site data. The planner's job is to determine the best way to decompose it into a partially ordered sequence of simpler single-site queries and generate the sequence of data moves that may be required before executing each of these single-site queries. In the case of a relational DDBMS, the optimizer is presented with a query that concerns a multiple number of tables. (Throughout the rest of this work a relational database model will be assumed, although the ideas presented here apply in a wider context. See [Date1975] for a description of the relational model.) The planner then tries to split the query into a set of simpler tasks (i.e., ones that deal with a smaller number of relations). The number of pieces into which the query can be broken, and the order in which the planner decides to execute them, may have a tremendous impact on the speed of execution of the multi-site query. The algorithms for query decomposition are usually heuristic in nature (see [Wong1976, Youssefi1978] for more details). Once the optimizer has broken up the multi-site query, it needs to assign the single-site queries to different processors and to decide which relations (or pieces of relations) to move in order to execute the queries. Thus, the DDBMS has to answer the following interrelated questions: how to break up complex queries into simpler ones, which data to transfer over the communication mechanism, and in which node(s) to do the processing.

There are many factors involved in making the above decisions. Typically, a query planner will try to minimize a given function of the resource demands for a query (the *cost function*). Some of those demands are the number of messages sent, the amount of I/O generated, and the CPU time utilized. For example, in R* [Williams1982], the optimizer tries to minimize a weighted sum of the three resources just mentioned. It is by modifying the cost functions used by the planners that load balancing will be implemented. It is worth mentioning that none of the currently implemented or proposed systems has made use of load balancing as a means to improve the performance of the DDBMS. This is unfortunate, especially since much of the earlier work in DDBMS optimizers has made assumptions about processor loads which were often unrealistic. Some of the simplifications have been: considering the available processors as identical; assuming that differences in load among systems can be disregarded; focusing on transmission delays as the primary cost factor. The last simplification may be valid in some point-to-point networks like the ARPANET [McQuillan1977], but seems incorrect in a local area network [Clark1978] environment. The first two simplifications essentially assume that a given task will generate the same amount of work in whatever processor it is executed. This neglects the queuing effects present in all systems (i.e., effects like thrashing [Denning1968] in virtual memory systems) that we are familiar with: in a heavily loaded system, a small job will degrade system performance far beyond its intrinsic weight. It is clear that, by considering the actual computational load of the available processors, one can attempt, through load leveling techniques, to improve the execution time of queries.

In the following chapters, the query optimizers for two running DDBMS's, R* and distributed INGRES [Stonebraker1977, Stonebraker1976], will be described in more detail.

1.3. The Problem

This study focuses on the following question: can the performance of a DDBMS be improved by using load balancing techniques? It is clear that some degree of improvement could always be achieved by giving more knowledge about the system to the query planner, but only if the gains are sizable will this proposal be of more than academic interest. Note that system designers, who have sunk many person-years of effort into their query optimizers, might be reluctant to discard their current software and develop new code that incorporates load balancing. Thus, it would be desirable to reduce to a minimum the changes the introduction of a load balancing scheme will require. 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. Hence, a better statement of the problem would be: can load balancing strategies be easily introduced into current optimizers in such a way that the performance of the system will be substantially improved in realistic situations of imbalance and under reasonable conditions?

1.4. Our Approach

In order to answer the questions posed in the previous section, the following approach was taken: First, some exploratory simulation studies were performed in order to get some initial insights into the problem; then, an experiment was run with an existing system, the R* DDBMS at the IBM Research Laboratory in San Jose; finally, a second simulation model was completed, this one of distributed INGRES, which was calibrated by measurements performed on existing software. It is hoped that, by utilizing two different techniques, measurement and simulation, and by looking at two very different systems (R* and distributed INGRES), the results of this work will be regarded as having both greater reliability and wider applicability than if they were only based on a single system and a single evaluation technique.

1.5. Outline of the Thesis

This thesis consists of five more chapters. The next surveys the existing load balancing literature, and is followed by a chapter on the preliminary simulation modeling study. Chapters 4 and 5 describe the work performed on R* and on distributed INGRES, respectively. The final chapter presents the conclusions that can be drawn form the results, and outlines our plans for future research.

CHAPTER 2

Survey of Load Balancing Strategies

In this chapter, some of the previous work in the area of load balancing is summarized. Since the load balancing literature is extensive, not all of the papers that have been published on the subject are described here. However, the most widely used approaches are discussed in some detail, and references are given to a large number of studies. Most of the work surveyed has been found to be somewhat inappropriate for practical implementations, due to the restrictive nature of the assumptions made by the authors. Some actual implementations do exist, and they are also surveyed.

There are various ways to categorize the available literature: on the basis of the system configuration and topology assumed by the study (two processors vs. an arbitrary number of processors, broadcast vs. point-to-point network, heterogeneous vs. homogeneous nodes), of the preemptive vs. non-preemptive nature of the policies, of the type of solution proposed (optimal vs. heuristic), of the solution method (minimal cut-maximal flow, dynamic programming, queuing networks, simulations), and so on. We chose to structure this survey on the basis of the observation that, in general, one of two somewhat related problems motivates a research effort in the load balancing area: either task assignment or job routing. The first involves breaking up a process into multiple subunits (tasks), which will run in parallel in a distributed system. The second entails scheduling the (usually independent) jobs that enter (at possibly different places) a distributed system. As we describe more fully in Section 3.1, it is the job routing approach that is of most interest for our work.

Task assignment policies try to achieve maximal parallelism, while taking interprocessor communication (IPC) delays, and maybe other constraints (such as memory size), into account. Papers on task assignment typically assume that all tasks are available at the same time, and that their running times, as well as their IPC patterns, are known in advance. These assumptions do not hold in general for distributed systems, but are partially justified in the context of those studies for the following two reasons: (1) since a single process is being broken up into many tasks, all the pieces are ready to be scheduled at the same time, and (2) presumably the program has been sufficiently studied before it is divided into subunits that the execution time behavior of the tasks can be predicted. One major flaw of these approaches is that, in general, they do not reflect certain non-linear effects such as multiprogramming interference. On the positive side, this kind of work results in optimal policies for the models considered. Task assignment models have been studied by three different approaches: by graph theoretical techniques, by dynamic programming, and by integer 0-1 programming. The first three sections of

this chapter deal with these topics.

In job routing, the goal is to offload overutilized resources in a system with, usually, excess capacity, by balancing requests among the available resources. Much of the work in this area is similar in flavor to the queue scheduling problem, and indeed some of the approaches have consisted of modeling a distributed system as a set of queues with no interprocessor communication overhead or delays modeled. This is the subject of the fourth section of the chapter. The fifth section describes further work in job routing, where a load balancing decision is made based on some information about the global state of the system, this information usually being broadcast by the processors involved. Next, the best known existing implementations of load balancing are described. The following section consists of a brief guide to work not surveyed in previous sections. The chapter concludes by presenting a summary of our comments on the various approaches.

2.1. Task Assignment: Graph Theoretic Methods

The first work in this area was done by Stone [Stone1977], whose studies were motivated by the processor-satellite installation described in [Michel1976]. Stone first considered the two processor case. Both tasks and processors are drawn as nodes in a graph; a link appears between two tasks if they communicate, and the link's weight is the communication cost. A link is created between each processor and all the tasks; each link is labeled with the cost of executing the task at the other processor. A maximal flow-minimal cost cut in this graph represents an optimal assignment. In [Stone1977] Stone also was able to generalize this approach to the n processor case. However, there are no efficient methods to solve the n-way minimal cut problem; hence, the algorithm can only be used as an off-line processing technique. There are other deficiencies in this model, such as the absence of any task ordering assumption, and the need for much information about the tasks to be available. In a later paper [Stone1978], Stone presented the following existence proof: for the two processor case, there exists a load metric (which he called the *critical load parameter)* such that, if a module is assigned to the "less loaded" processor (as defined by the critical load parameter), as the load increases in the more loaded processor, there is at least one optimal assignment that also schedules that module in the less loaded processor. In essence this means that a stable policy was found (since, as the load becomes more imbalanced, previous decisions do not have to be revised). Rao *et al.* [Rao1979] extended the graph-theoretic method to account for memory limitations in one of the two processors. Wu and Liu [Wu1980] considered more complicated communication costs (i.e., the IPC costs are related to the distance separating the processors) while assuming that the computational costs were identical for all machines. Bokhari [Bokhari1979] extended Stone's model by adding two new factors: (1) the cost of reassigning tasks, and (2) the cost of task residence. Lo and Liu [Lo1981] considered a group of near optimal heuristics to solve the maximal flow-minimal cost problem for n processors. For a more detailed survey of the graphical method see [Chu1980], where a program flow method to estimate IPC

costs is also described.

2.2. Task Assignment: Dynamic Programming Approaches

For the special class of tree-structured programs, Bokhari [Bokhari1981] was able to find an $O(mn^2)$ algorithm for scheduling m tasks among n processors. He first considered scheduling in a system where computational costs are machine dependent, but time-invariant (his model includes IPC costs as well). Bokhari's approach consists of constructing a graph, where the nodes are labeled with a tuple (i,j), representing the assignment of task i to processor j. The link between nodes (i,j) and (k,l) has a weight proportional to the IPC cost between tasks i and k (when they are assigned to processors j and l, respectively), plus the cost of running task k at processor l. The shortest path between the beginning and the end of the graph represents the optimal assignment, and can be computed by a dynamic programming algorithm. Next, he considers scheduling across time, i.e., he assumes that the computational costs are time varying, and one can delay a task so that it runs when a CPU is less loaded, but there are deadlines that must be met, or a penalty paid. This problem turns out to be solvable by essentially the same method as the previous one. Although this extension is more general than the approaches in the previous section, since computational costs can change with time, there is still no way to model the increased delays due to many tasks being assigned to the same processor.

2.3. Task Assignment: Integer 0-1 Programming

Chu *et al.* [Chu1980] described a model which can be resolved by an integer programming technique. Their approach consists of setting up a cost equation and some constraint equations; tasks are assigned to minimize the cost equation, while still satisfying the requirements of the constraint equations. The cost equation has terms of the form $a_{i,j}*c_{i,j}$, where $a_{i,j}$ is 1 if task i is assigned to processor j, and $c_{i,j}$ is the cost of running task i at processor j; for a given task assignment, the sum of these terms equals the total cost of the assignment. The equations can be set up to include not only these computational costs, but also the costs due to IPC, and to model various constraints, such as real time deadlines, limited memory and task ordering. This method requires the availability of extensive workload information and is computationally expensive as well. Chu *et al.* mention that an assignment problem with 15 processors and 25 tasks would generate 8000 constraints and 2500 unknowns, and they cite a study that found that the required calculations would take on the order of a few minutes on a CDC 6000 series machine. While this is appropriate as an off-line method, it is unsuitable for a runtime approach. Ma, Lee and Tsuchiya [Ma1982] explored branch-and-bound solution methods to the 0-1 integer programming problem. Although their approaches were more efficient, they still required about a second of CPU time on a CDC 6600. It should also be mentioned that a 0-1 integer programming approach was taken in [Ceri1982] for optimal query processing in DDBMS's. Ceri and Pelagatti focused on data transfer costs

for join operations and neglected load considerations. However, since these authors claim that for queries with a small number of joins their approach is suitable for implementation, perhaps it would be worthwhile to extend their model to include load balancing considerations.

2.4. Job Routing: Queuing Network Approaches

Chow and Kohler [Chow1977] first studied the job routing problem by examining the two homogeneous processors case. They compared the turn-around times of jobs using various queuing models. A recursive solution technique was used to solve the equations describing a distributed system model composed of two queues and a communication channel, the latter being used for load balancing. When the load is unbalanced, jobs are preempted, and the channel is used to reassign the jobs from the more loaded processor to the other one. The models studied included: (1) two M/M/1 systems (no load balancing at all), (2) same as the first, but arriving jobs join the shortest queue, (3) like the second, but with the communication channel, (4) two M/M/1 queues with the communication channel (i.e., in the fourth model, jobs initially join a queue randomly, whereas in the third model they arrive at the shorter queue first), (5) an M/M/2 system and (6) a single queue with twice the arrival and departure rates of the other cases. The authors show that the turn-around times degrade from model (1) to (6), with the behavior of model (4) depending on the channel speed. Chow and Kohler extended their results in [Chow1979] to include multiple heterogeneous processors. They compared a non-deterministic policy, where jobs were assigned to processors with some fixed probability proportional to processing power, with three deterministic policies, where the routing decision was based on the current state of the system. The goals of the three deterministic policies were: minimum response time (for the job currently being scheduled), minimum system time (the time for all the jobs in the system to finish), and maximum throughput (maximize the system throughput, but only for the interval between the current time and the time the next job is expected to arrive at the system). Recursive techniques were used to find approximate solutions for the deterministic policies (for the two processor case only). The authors found that maximum throughput was the best policy from the viewpoint of achieving minimum mean job turn-around time; this was ascribed to the fact that the maximum throughput policy takes arrival rates into account. Kohler and Chow could not extend their approach to more than two processors. Ni and Hwang [Ni1981] found that the non-deterministic policy suggested by Chow and Kohler was non-optimal. In the same paper, Ni and Hwang also explored load balancing in networks with multiple job classes.

2.5. Job Routing: Load Broadcasting Methods

Various authors have approached the load balancing problem by considering a network where nodes are fairly autonomous, and jobs can enter the system at various places. Decentralized load balancing schemes are suitable for such environments, and

the policies that have been suggested involve making routing decisions based on some information about the loads existing elsewhere in the network. This load information is broadcast by each processor to all or some of the other nodes in the system. Bryant and Finkel [Bryant1981] considered a point-to-point network composed of homogeneous nodes. Their load balancing strategy consists of using load estimates to form pairs of processors that differ widely in load. The more loaded processor in a pair then migrates some jobs to the other. Jobs are chosen to be migrated if their estimated remaining running time is greater in their present processor than in the less loaded one, even when taking the time required for migration into account. Possible differences in file access costs are neglected. Four policies for compiling the running time estimates were studied, as well as two variations of the pairing scheme. A similar algorithm was studied by Krueger and Finkel [Krueger1984]. They described the *Above Average* strategy, which is as follows: when a node thinks that its current load is "above average" (by a given threshold amount), it broadcasts that fact, and processors less loaded than the average respond. If no processors respond, the querying processor informs the others that a new average load estimate is needed. Similarly, if, in a given interval, no node broadcasts that it is overloaded, the consensus average load must be revised downwards. The strategy was studied by simulations, which showed that it improves the mean process response time, without significantly overloading the communication channel. Livny and Melman [Livny1982] have also studied load balancing in broadcast CSMA networks by the use of simulations. They explored three different policies: (1) state broadcast, (2) broadcast when idle, and (3) poll when idle. It was found that, as the network grows larger, the mean response time of the jobs decreases, until the effects of contention on the communication channel begin to be felt, and then the mean response time increases. Depending on a variety of system parameters, any of the three policies investigated could be the best. However, in all cases, sizable improvements over the no load balancing case were found. More details about this work, as well as further work in the context of point-to-point networks, can be found in [Livny1983]. Chuanshan, Liu and Railey [Chuanshan1984] have studied load balancing algorithms that use load broadcasting in networks of homogeneous processors.

2.6. Job Routing: Implementations

There have been a few attempts at implementing load balancing in distributed systems. The Purdue implementation, *rxe* [Hwang1982], consists of modifying extensively used commands that are CPU intensive (such as compilations or text processing), so that, when a user runs the modified version of the commands, the work is done on the least loaded machine. The system keeps track of various measurements (such as swapping rate, memory contention, number of processes, and so on), to determine the least loaded machine. At Berkeley, a distributed shell, *dsh* [Presotto1982], was implemented under Berkeley UNIX 4.2 BSD [Leffler1983a]. In this approach, there are server processes at the available processors that keep track of every CPU's load average (defined in 4.2 BSD as an exponentially smoothed estimate of the number of jobs in the

run queue during the last minute), to determine the least loaded machine. A user submits a job to dsh, and the program decides where to run it. Only non-interactive jobs can be handled by the system, and the lack of a distributed file system makes load balancing less effective than would be possible. The implementation is not very efficient: it takes several seconds to transmit even a small job, with no data, to another machine over a ten megabit Ethernet [Metcalfe1976], using Berkeley's TCP/IP implementation [Postel1980b, Postel1980a, Leffler1983b, Leffler1983c]. At Brown University, a main processor-satellite system [Michel1976] was built for graphics research. All the modules composing a program were compiled in both systems. Load balancing is implemented by having the flow of control of a program switch between the version of the code in the main processor and the equivalent version in the satellite, depending on which of the two processors has the least load. Adesse Corporation [McGrath1983] has designed a load balancing mechanism for IBM VM systems. Their software assigns incoming jobs to the processor with the least amount of load.

While there have been relatively few implementations in the past, many of the current research projects in distributed computing intend to explore the subject. For example, the designers of the Crystal [Cook1983, Cook1983] project at the University of Wisconsin-Madison will soon experiment with a load balancing scheme [Finkel1984]. The designers of LOCUS [Popek1981] at UCLA have built into their system many of the capabilities needed for a load balancing scheme (i.e., transparent remote execution and a distributed file system). While they have not currently explored different load balancing strategies, they are interested in doing so in the future [Popek1984]. Finally, the designers of Berkeley UNIX 4.2 BSD are actively involved in load balancing research for UNIX-oriented distributed systems running in CSMA-CD local area networks [Ferrari1983].

2.7. Other Work

There has been much work in load balancing not covered by the classification scheme of this chapter, and only brief surveys of the work done in each particular area were presented in the previous sections. In this section some further pointers to the literature are provided.

Coffman *et al.* [Coffman1977] describe applications of bin-packing to load balancing. Chou and Abraham [Chou1982] have described an approach based on Markov decision theory. Jain [Jain1978] details a control theory methodology for the problem. Levy [Levy1982] gives an example of the benefits of load balancing for a logic and timing simulation application running in a network of VAXes and a Cray-1. Stankovic [Stankovic1981, Stankovic1983] has discussed approaches to load balancing based on Bayesian decision theory. Kratzer and Hammerstrom [Kratzer1980] developed a cost model of load balancing which was used to show that, under their assumptions, the problem is NP-complete. Tantawi and Towsley [Tantawi1984] use product-form queuing networks to model a distributed system, and consider two algorithms for finding an optimal load

balancing strategy.

2.8. Summary

Many of the papers surveyed in this chapter deal with optimal or near optimal solutions to a model of load balancing. However, the deficiencies of the models involved make those approaches less than fruitful from an implementation point of view. The heuristic approaches do not seem to come from a detailed and systematic study of the behavior of processes in a distributed system, but rather from some intuition about what such processes should be doing. A further point is that, although many authors consider preemptive schemes, practically no existing systems support process migration; the only implementations of which we have knowledge are [Powell1983] and [Popek1981]. There is much work to be done in this area to develop policies that are implementable and perform well, but that, at the same time, rely on a firm theoretical foundation. Although the work described in this thesis does not directly deal with the question of how best to implement load balancing, the methods presented in the following chapters should shed some light on the subject, at least for the case of distributed database systems.

CHAPTER 3

The Preliminary Simulation Model

As described in the introductory chapter, a need was felt for an exploratory set of simulations, which could be used to gain some insight into the problem of integrating load balancing strategies into DDBMS's. In this chapter, those experiments and their results are described.

3.1. Introduction

In the literature survey chapter, two different conceptual views of load balancing were described: task assignment and job routing. In the context of DDBMS's, the former approach would be appropriate under either of two conditions: (1) if the problem of interest were to allocate among the network nodes the different subqueries arising from a single complex query, or (2) if one desired to schedule all the queries from a single high level language program. However, if instead we were interested in scheduling *all* the queries arriving at a DDBMS, job routing would be the correct framework. In a DDBMS the workload is actually composed of independent requests, which arrive at different sites in the network, and each request may be a complex one. Hence, both load balancing approaches may be of interest to DDBMS designers: one would focus on system-wide concerns, the other on a particular user's needs. However, since we were interested in global scheduling strategies, the job routing viewpoint was taken in the simulations described in this chapter (as well as in the rest of this dissertation).

Since some of the load balancing strategies to be explored led to models which were too complex to be analytically tractable, we chose to use simulation models. Our models were constructed using SIMPAS [Bryant, Bryant1980], a simulation package developed at the University of Wisconsin at Madison. SIMPAS enables the construction of event-driven simulations by providing a pre-processor that translates the simulation language statements into PASCAL commands.

The model used in the experiments is a simple one, mainly because the simulation results are not meant to be accurately predictive, but rather to reflect broad differences in performance if they exist. A second reason for this decision is that a complex model, when its many parameters cannot be measured or validated, is less useful than a more tractable and simple one. Other reasons for avoiding complexity were the desire to run relatively inexpensive simulations, and the lack of comprehensive workload information. It should also be pointed out that the model used is fairly general, and is not meant to represent any particular DDBMS, but rather to reflect the most relevant features of those systems. For this reason, the results presented in this chapter should have wide

applicability.

The second section of this chapter describes the model used for the experiments. The next two sections detail the two experiments performed: the first simulation tries to gauge the benefits of load balancing, and the other experiment focuses on the frequency requirements of load information updates. The last section presents some closing remarks.

3.2. The Model

For this thesis, the environment of interest is one composed of a number of heterogeneous processors (i.e., processors that differ in architecture, processing power and in the attached I/O devices), connected by a local area network. We will study a system where users demand some service from the DDBMS and then exit. Thus, since users leave the system after their requests are met, an open simulation model is used.

In the experiments, each processor was simulated by a central server model of a computer system [Ferrari1978]. The model of the processor used is shown in Figure 3.1. The jobs[1] arrive with an exponential inter-arrival rate. They may come into the system at any processor, and are sent to the local CPU queue if there is no load balancing; otherwise, the jobs are sent wherever the load balancing algorithm chooses. Processes can then request disk or terminal I/O. If jobs exhaust their CPU time slice, they are rescheduled. Processes are also able to send messages to other machines on the network. The most interesting actions jobs can perform are to request remote data (READ), or to WRITE information to a non-local portion of the database. This is modeled by sending message tokens to all the sites that have the data to be read or written, and by making the requesting job wait for acknowledgements from the other sites involved. When a message token arrives at its destination, it generates a new job that will perform locally the appropriate I/O. When that remote job is finished, a new message is sent back to the original site, containing either the desired data (in the case of a READ), or an acknowledgement (if a WRITE was initiated). When the messages from all the remote sites involved in the operation return to the original site, the waiting job is rescheduled. Finally, as jobs terminate their actions, they are taken off the system. In the simulation runs described in this chapter, jobs did not perform terminal I/O, or locally initiated disk I/O, except for the local disk I/O resulting from "remote" I/O requests (whose destination could be any site, including the local one).

The DDBMS was modeled as a collection of the processors described in the previous paragraph, all interconnected via an Ethernet network. The Ethernet was represented as a queue with an exponential service time. Ethernet models like the one developed by Almes and Lazowska [Almes1979] could have been used, but they were felt to be unnecessarily complex for these simple simulations (especially given the light loads

[1] In this chapter, we denote as "queries" the actual database interactions (the requests typically expressed by query language statements) and by "jobs" we mean the execution of a high level language program with embedded query language requests

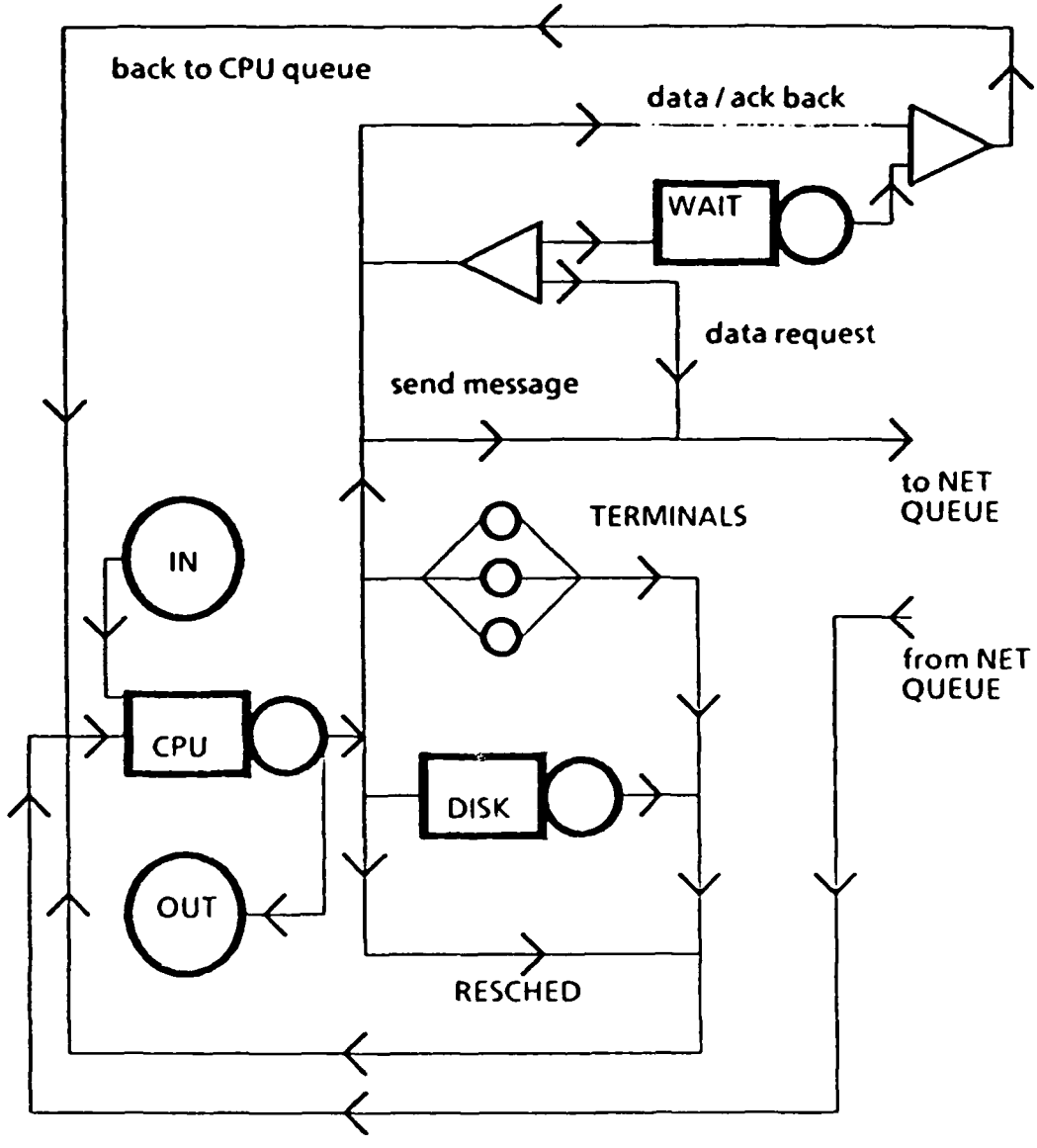


Figure 3.1: Model of a Single Processor

present on the Ethernet in the model). Figure 3.2 shows the model of the distributed database network.

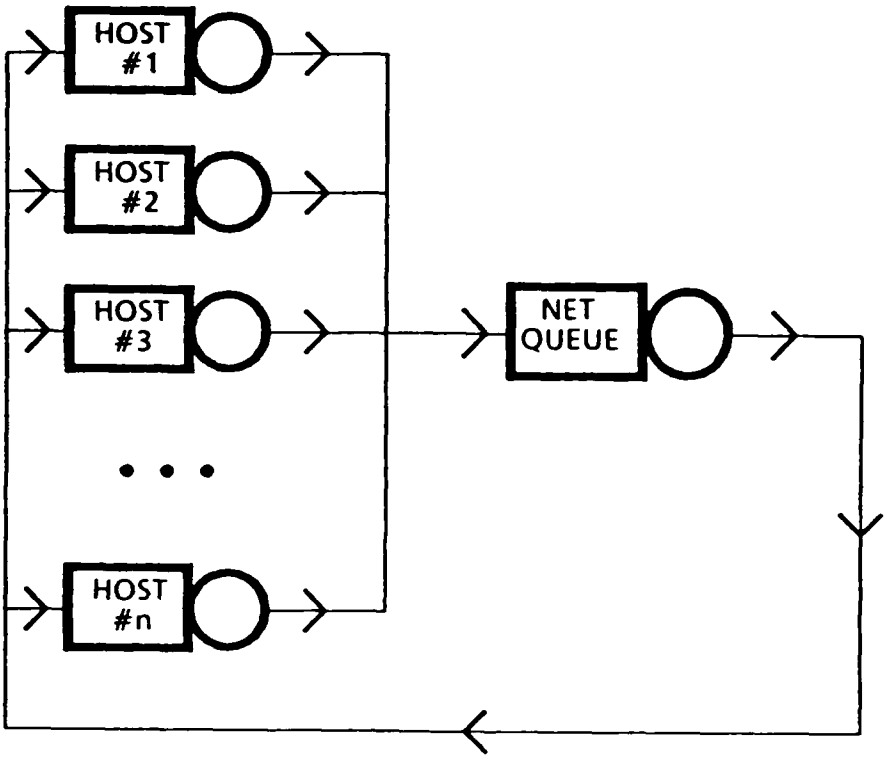


Figure 3.2: Model of the DDBMS

The parameters used in the model are shown in Figure 3.3, together with their values. These values were determined from measurements previously made in U.C. Berkeley's computing environment. The network response time was measured by timing the TCP/IP implementation in Berkeley UNIX 4.2 BSD. Communications between two processes were benchmarked [Hunter198 1]; each process was running on a separate DEC VAX 11/750, and communicated over a 3 megabits/s Ethernet (which, because of software overhead, had an effective bandwidth of about 1 megabits). We modeled relatively simple jobs, each requiring about 50,000 VAX instructions to complete; we felt

this to be a reasonable number of instructions to access and process a page or two of database information. Each remote read (or write) resulted in a fixed-size 256 byte message being sent across the network. The values of the other parameters were measured, or estimated, in a previous modeling study of DEC VAX 11/780's running at Berkeley [Alonso1982].

Simulation runs were performed that lasted from 10000 to 100000 milliseconds of simulation time; it was found that a simulation time of 50000 milliseconds was sufficiently long for the simulation results to converge (by this we mean that the values of the performance indices used varied by less than 10% between runs as the simulation length was increased past 50000 milliseconds). This is the simulation length used in the experiments reported in this chapter.

parameter	value
Nodes in network	5
Network service time	1 Mbit/s
User think time	1250 ms
Disk service time (11/780)	30 ms
Disk service time (11/750)	50 ms
CPU service time (11/780)	10 ms
CPU service time (11/750)	15 ms
CPU quanta used	5
Branching probabilities:	
rescheduling	.8
remote I/O	.2
Data sent per remote I/O	256 bytes

Figure 3.3: Parameters for the Simulation Model

3.3. Preliminary Experiment #1: the benefits of load balancing

The goal of this experiment was to gain an appreciation for the possible benefits of load balancing. This issue was explored by examining the effect of load balancing on two performance indices: mean job turn-around time and mean system throughput. The first index is defined as the mean time elapsed between job initiation and job completion. The system throughput is defined here as the total number of jobs completed per second by the system during the simulation run.

The simulations were performed for a variety of job arrival rates, both with and without load balancing. The values of both indices were measured under two scheduling algorithms. Specifically, the two scheduling policies compared were: (1) incoming jobs run at the node at which they entered the system, and (2) new jobs are routed to "less loaded" processors (as defined by the load balancing strategy). The load metric used for each machine was defined to be the number of jobs waiting in the CPU queue (i.e., the load average defined in a previous chapter), which is similar to the metric displayed by the **w** command in Berkeley UNIX 4.2 BSD. The load balancing policy consisted of sending the incoming jobs to the processor with the smallest value of the load metric (even if the difference between the remote and local host was small). No preemption was modeled (i.e., once a process was assigned to a node, it was never migrated again), mostly because few existing systems support job migration (due to the complexity of the implementation), and this study focuses on easily implementable alternatives.

The results of this simulation were expected to show optimistic improvements in performance while using load balancing. This was so because, although the overhead of moving a job was modeled, two optimistic simplifications were made: (1) the cost of gathering and sharing the load information was not accounted for, and (2) it was assumed that this information was always up to date. The first assumption can be justified for many systems, since load information is routinely shared in many local area network based systems. For example, in Berkeley UNIX, load information is shared among the network hosts, and can be displayed via the **ruptime** command [Leffler1983a]. This command is implemented by creating daemons (**rwhod** daemons) in all appropriate processors, which will periodically send each other the value of the load existing in their local systems (as defined by the **w** command previously mentioned). Since processors incur this overhead regardless of whether the load is being automatically balanced, the cost of information sharing can be neglected. The second assumption is more questionable; stale load data can lead to performance degradations. This issue will be addressed more fully in the next section.

The simulation whose results are described in this section models a network of five hosts: three DEC VAX 11/750's and two VAX 11/780's (see Figure 3.4). The five systems are joined by a 3 Mbits/s Ethernet. Jobs arrive at each of the three VAX 11/750's with equal probabilities. None originated at the VAX 11/780's; however, since jobs required data from all systems with equal probabilities, some I/O work has to be done at the 11/780's. When there is no load balancing, there is no other work performed by

the 11/780's. When the load is being balanced, some of the jobs that originate at the 11/750's may be routed to one of the 11/780's (as well as to a less loaded 11/750).

The mean arrival rate at each 11/750 of incoming DDBMS jobs was varied from 2 to 33 jobs per second (thus, the total system arrival rate changed from 6 to 99 jobs per second). Figure 3.5 shows the effect of load balancing on the average job turn-around time, for a variety of arrival rates. The throughput achieved (by the entire system) in the simulations is depicted in Figure 3.6. As can be seen, load balancing dramatically

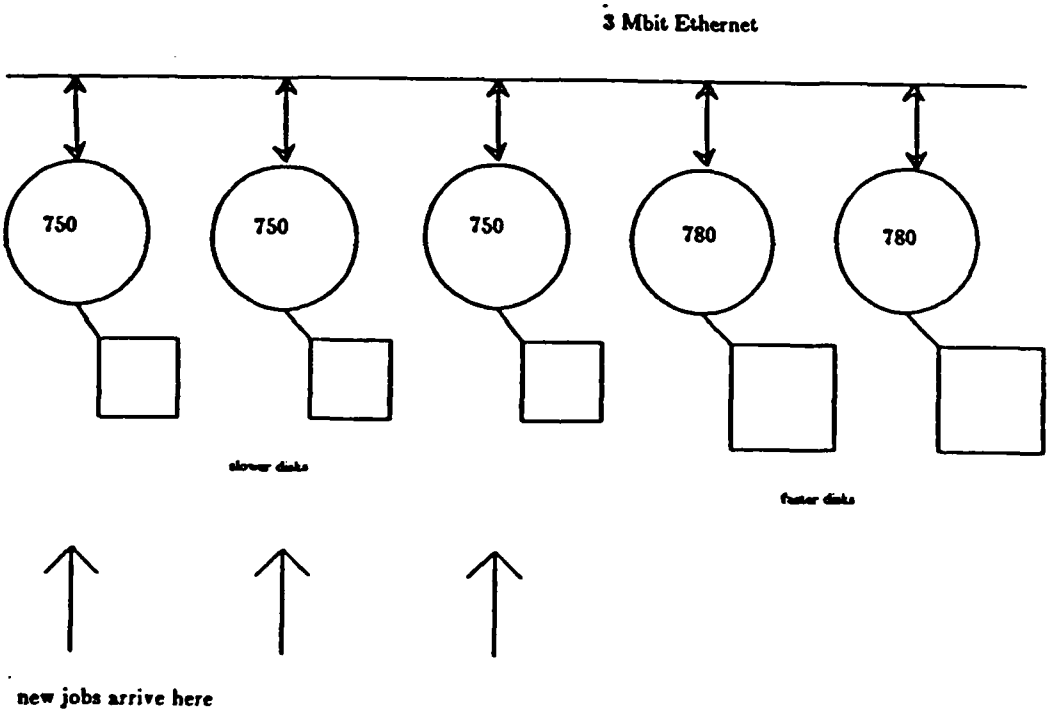


Figure 3.4: Model Used in Experiment # 1

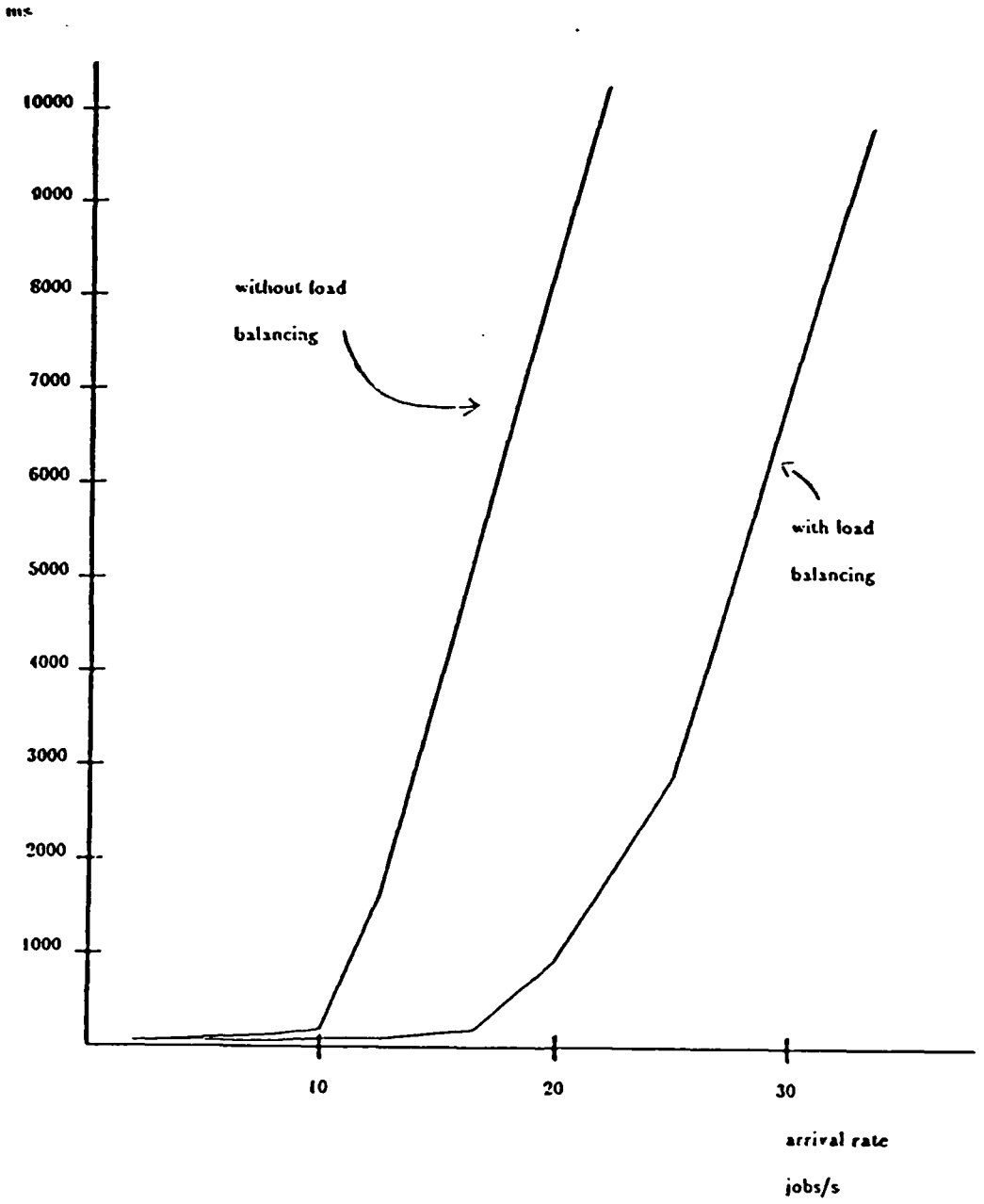


Figure 3.5: Turnaround Time vs. Arrival Rate - Experiment # 1

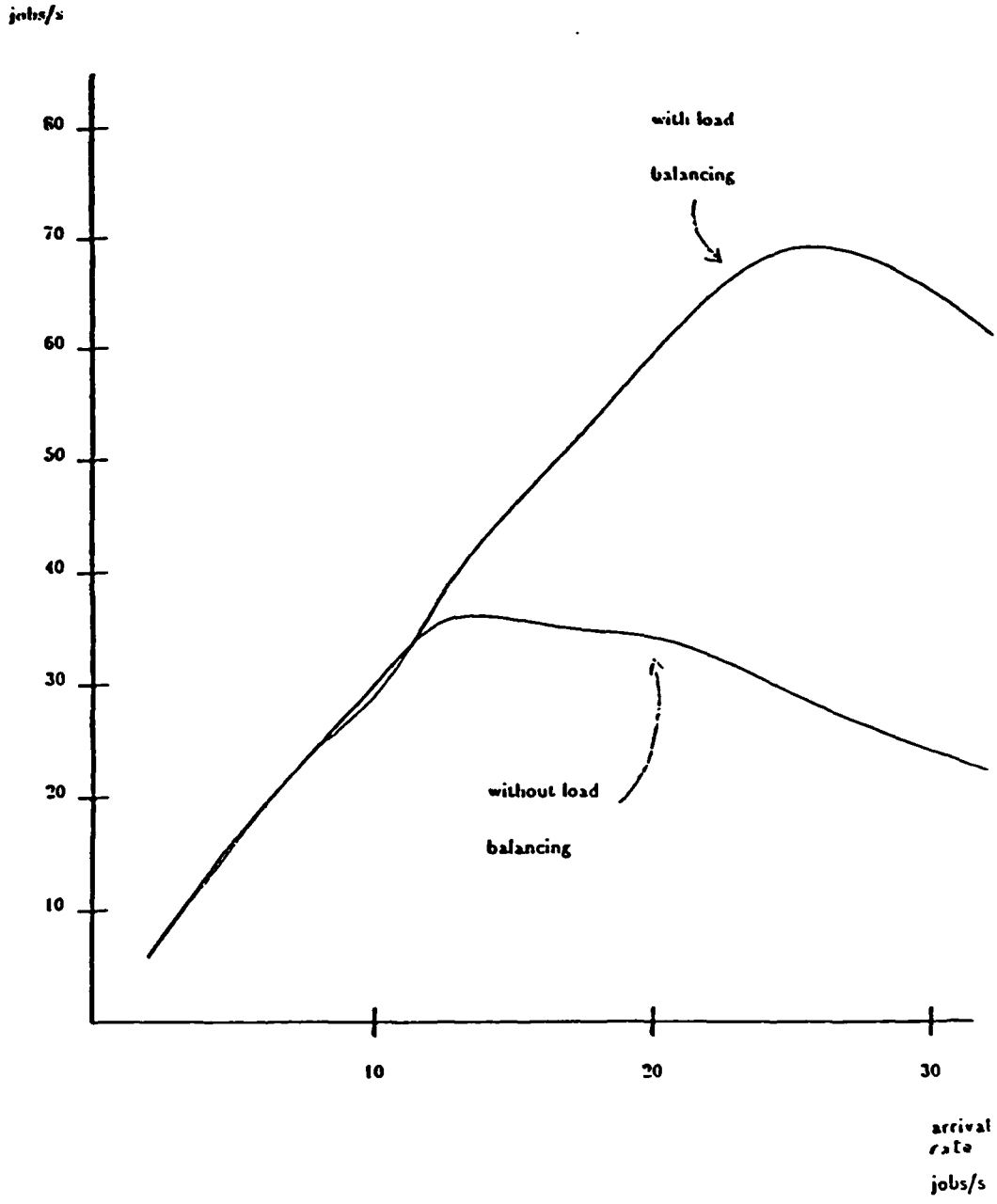


Figure 3.6: Throughput vs. Arrival Rate - Experiment # 1

improved both job turn-around time and the system throughput. Clearly, the network bandwidth is high enough that, for the simulated conditions, the overhead of transferring jobs to less loaded processors is dominated by the running time improvement due to load balancing. As a matter of fact, in the simulations performed, network utilization never even reached 40 per cent.

It should again be pointed out that the actual improvements are somewhat optimistic. When the results of actual experiments performed on real systems are described in later chapters, the effects of load balancing will be seen to be less dramatic, but the benefits obtained to be still sizable.

One final comment needs to be made about the load balancing strategy. Even though we have stated that we are not concerned with finding the "best" load balancing scheme, one may wonder as to the effectiveness of the one we employed here. In order to address this issue, the *load imbalance* factor was measured. This factor is the standard deviation of the (time average) mean values of the load metric for each of the nodes in the network. The load imbalance factor is meant to reflect how well a strategy is balancing the load (that is, the "load" as defined by the load metric). In this case, the load imbalance is defined as the standard deviation of the mean lengths of the CPU queues in all the hosts. For all the values of the arrival rate employed, the load imbalance factor was small when load balancing was applied; it was less than 3.0 for the fastest arrival rate, and less than 1.0 for all the others, signifying that the load balancing strategy was being successful in its aim. This would indicate that the load average, as defined in UNIX (see Chapter 2), would be a likely candidate for a load metric in a load balancing implementation; it is both simple to use and inexpensive to implement.

3.4. Preliminary Experiment #2: update frequency of load information

As was mentioned in the previous section, a flaw in the first experiment was the assumption that up-to-date load information is always available to all machines. In order to examine load balancing in a more realistic light, load sharing with an "imperfect" load metric must be considered. In this section, the effects of stale data on the performance of the load balancing strategy are explored. The same model (see Figure 3.4) and parameters (see Figure 3.3) employed in the previous experiment are used, but now the load information (i.e., the values of the load metric) lags behind the actual values. This is modeled by having the system periodically refresh each processor's estimates of the load existing at other nodes. For a given mean job arrival rate, as the time between load information updates increases, more jobs arrive during that interval, and more assignment decisions must be made, with data that becomes increasingly out of date with each decision. Hence, the system may be making wrong decisions for more and more jobs, and the benefits of load balancing may decrease. In particular, we would expect that the following unstable situation might arise: the system routes all new jobs to a site that was idle (relative to the other processors) the last time that load statistics were broadcast; that site becomes overloaded (while all other sites have little to do) and.

at the next broadcast, some other processor is chosen as the "victim" for the duration of that update interval. Thus, the system is constantly changing its choice of victim and the effective throughput of the entire system is scarcely more than that of a single host.

For this experiment, the mean job arrival rate was kept fixed at 12.5 jobs per second for each of the three 11/750's; no jobs entered the system at the two 11/780's. The update interval was varied from 0 to 5000 milliseconds. Figure 3.7 shows the effect of decreasing the refresh rate on the mean turn-around time experienced by jobs in the system. The abscissa of the graph is labeled in two ways: by the update interval, and by the number of jobs that entered each machine during the update interval (the latter quantity equals the update interval times the machine job arrival rate; the total number of jobs that entered the system is three times that amount). As would be expected, when the update interval increases, the mean turn-around time degrades. In order to understand the significance of the graph, an additional line is shown in the figure. The hatched line marks the mean turn-around time for jobs in the system if no load balancing takes place. As is evident from the graph, when the update interval increases, the performance of the system begins to degrade from its best result (with perfect information), until, at around 1500 ms, the mean turn-around time is worse with load balancing than without it. Figure 3.8 shows that a similar result holds for the system throughput, although at a somewhat higher threshold (an update interval of about 2800 ms).

The results of this experiment show that, in order to implement a load balancing scheme, load information must be broadcast at a sufficiently rapid rate. This rate depends on the number (and type) of incoming jobs; for a system similar to the one being modeled, the refresh rate must be rapid enough that less than about 56 jobs enter the system during the update interval. Since the update rate must be fairly rapid, it means that one must look to inexpensive communication mechanisms to broadcast load information. For example, in running early versions of the **rwhod** daemons, the overhead involved in sending and receiving system load information was high enough that the update interval was increased from one minute (in the initial implementation) to three minutes. For a medium-sized network of single user workstations, that interval may be small enough, but certainly there are systems where a sufficient number of jobs enter during a three minute period that load broadcasting must be undertaken at more frequent intervals. In order to implement load balancing in such systems, various alternatives exist: (1) increase the update rate, (2) limit the number of processors involved in load balancing (so that communication overhead is smaller), (3) look for policies that are more stable in the face of uncertainty, (4) use a faster communication mechanism. Regarding the last alternative, it should be pointed out that the early implementation of **rwhod** mentioned above (whose overhead motivated the extension of the update interval) did not use the broadcasting characteristics of the Ethernet, but, rather, relied on each processor establishing a virtual circuit with every other processor (using the TCP protocol). Thus, the communication overhead was not proportional to the number of network nodes, but to the square of their number.

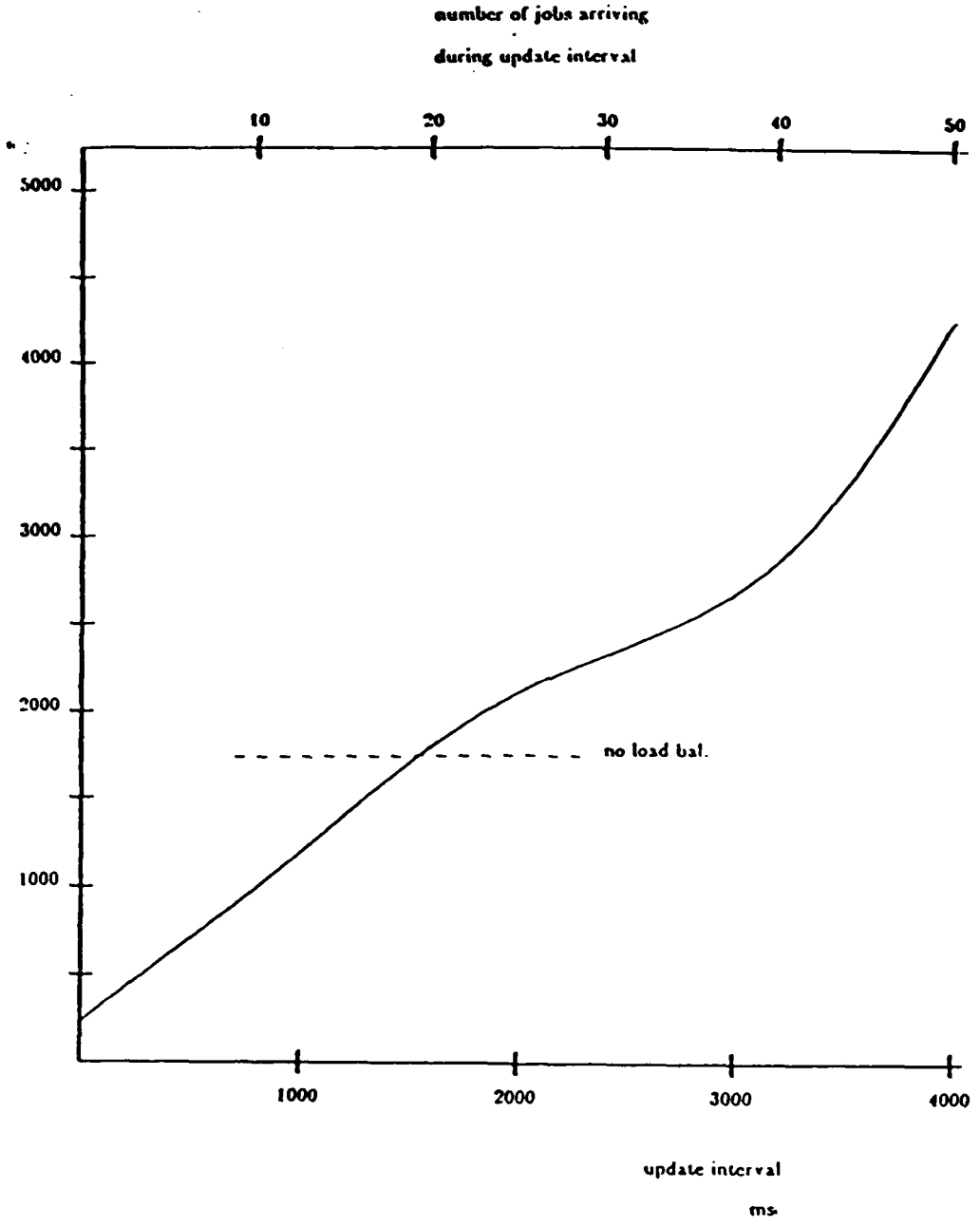


Figure 3.7: Turnaround Time vs. Update Rate - Experiment # 2

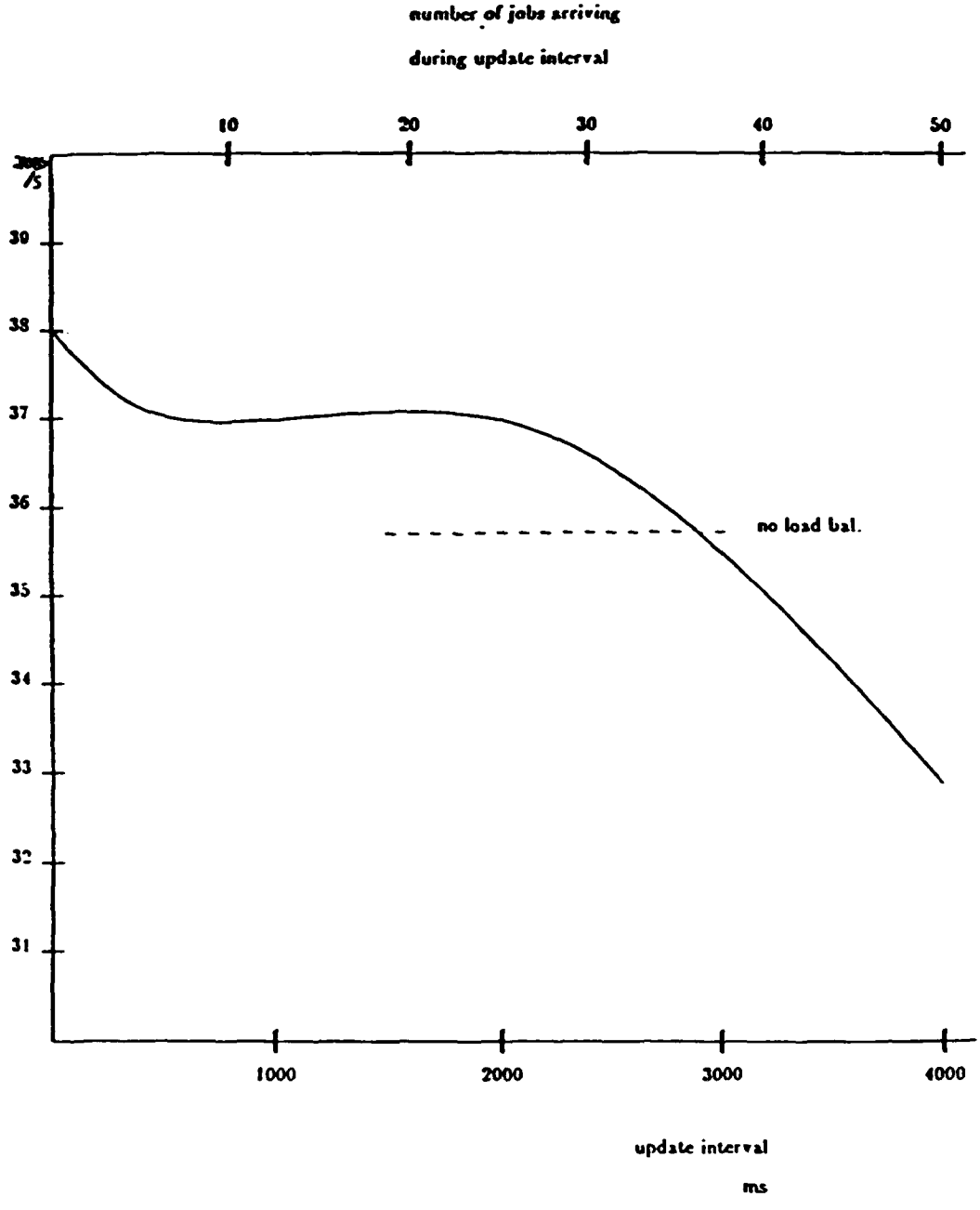


Figure 3.8: Throughput vs. Update Rate - Experiment # 2

Actually, selecting the appropriate size of the update interval in a real situation can be extremely complex. It is clear that the main factor determining the minimum frequency of updates is the dynamic behavior of loads in the system. In our model, where jobs arrived in a fairly static pattern (with a given distribution whose moments were fixed for the duration of the simulation), the total number of job arrivals during a given interval may be sufficient to characterize the behavior of the loads in the system. However, in more complex situations, where the job arrival patterns are not easily discerned, it is hard to make a more specific comment than that the loads must not change significantly during the duration of an update interval. To quantify this statement, it is necessary to consider the nature of the jobs entering the system (their resource demands, length of residence, and so on), the kind of processor statistics available (what they are, how accurate they are, how often they are gathered, and so on), and the specific load metric used, to name only a few of the obviously relevant factors. All this would require a serious study of load balancing mechanisms *per se*, which, as we stated previously, is not the central thrust of this thesis. However, such a study is a logical continuation point for studies in this area, and we hope to carry it out at a future date.

One final comment on this subject is that, although it is still not clear to us how often loads change in real systems, we have conducted some preliminary empirical studies of the values that the load average metric (described in a previous chapter) takes in 4.2 BSD systems. The results were encouraging in that the load average (of a single processor) seemed to change slowly (by plus or minus one) from one sample to the next (samples were taken every five seconds). All the large increases (say, increases of five or more) in the load average that were measured disappeared by the next sample (i.e., the load average would return to the value that it had before the large increase). We did not carry out sufficiently long measurements to feel confident enough to draw any strong conclusions from these observations.

3.5. Conclusions

This chapter was intended to present some of the preliminary results obtained by coarse simulations. As was seen, those results were indeed encouraging for load balancing. In the next chapter, we describe an experiment performed on a running system, the R* DDBMS developed at the IBM Research Laboratory in San Jose. Although the models used in this chapter were not designed to simulate R*, it will be seen that the benefits of load sharing suggested by the results of these simulations still hold true in a real system such as R*.

CHAPTER 4

The R* Experiments

After performing the simulation studies described in the previous chapter, we carried out a set of experiments on a running system. This allowed our ideas to be tested in the real world, and our simulation results to be validated. This chapter contains a description of the experiments performed and of their results.

4.1. Introduction

The experiments described in this chapter were carried out using the R* DDBMS developed at IBM's San Jose Research Laboratory. R* has been sufficiently discussed in the open literature [Williams1982] that we need not do so here in detail. However, commenting on a few aspects of the system is in order[1]. In this section we will give some background information about R*, describe its query processing strategy, and explain why we chose that particular system for our experiments. The rest of the chapter completes our description of the experiments: the next section gives the details of the experimental setup, the third section describes the experimental results obtained, the fourth explores the feasibility of actually adapting load balancing schemes to the R* optimizer, and the last presents our conclusions.

4.1.1. Background

The R* DDBMS is the result of efforts to extend to a distributed environment the functionality of System R [Astrahan1976, Blasgen1979], a single-site DBMS that had been previously developed at San Jose. Like System R, R* is a relational database system, and runs on IBM processors. R* currently presents two interfaces to its users. The first is an interactive front end that checks and executes user queries; this interface is meant to be used for on-line single queries (called "ad hoc" queries). The second is implemented by means of a pre-processor, which takes as its input high-level language programs with embedded user queries written in SQL [Chamberlin1974], a database query language. Both interfaces employ the same query compilation algorithms. It should also be pointed out that R* is an experimental prototype, and not an IBM product. Thus, the system is not entirely free of bugs, and all its performance goals have not yet been met. The current research effort involving R* at San Jose is expected to produce further refinements of the system.

---

[1] All the details of R* presented in this chapter were current as of Spring 1984

4.1.2. Query Processing in R*

When the system is presented with a query (through either of the interfaces), it attempts to produce a query execution plan. The actual mechanism used is quite complex, and has been detailed in [Selinger1979, Selinger1980] and [Daniels1982], but will be sketched here for completeness. The query optimizer first generates a large number of possible query plans, based on the semantics of the query. The candidate plans are compared at various stages, and the less attractive ones are pruned, until only one plan remains. It should be noted that the plan chosen may no longer be feasible at the time the query is executed, since the state of the database might have changed. For example, a plan might make use of a secondary index that is deleted between the time the query is compiled and the time it is executed. Hence, the system must be prepared to cancel a given plan at run time, and re-compile the query.

In order to compare the alternative plans, the optimizer must assess their respective costs. It does this by estimating the amount of system resources that will be utilized by each particular query plan. As was mentioned in Chapter 1, the query optimizer considers the utilization of the following three resources: the CPU, the communication channel, and the I/O sub-system. The optimizer tries to minimize the total system resource utilization of the query to be executed; for each plan, three estimates are computed: (1) how much time the query will spend in the CPU, (2) how long will it take to move the required data across the network, and (3) how much time will elapse while doing disk I/O. The optimizer then chooses the plan with the smallest sum of the estimated times. Note that the chosen plan might not have the smallest turn-around time, since the three activities considered above may occur at least partially in parallel.

4.1.3. Why R*

There were many reasons why R* was a particularly good choice for our experiments. The most important one is that R* has been implemented and is running, and thus our results carry more weight than if they had been obtained based solely on a paper design. Also, R* has an extremely complex query optimizer; if we can show that our ideas about load balancing can be easily implemented in such a system, we can be more confident that we can adapt them to less sophisticated DDBMS's. Furthermore, a lot of effort has been expended in the R* project on algorithms for compile time query optimization. When we selected R*, we hoped to show that runtime considerations can be at least as important to the performance of queries as compile time ones, and thus point out that, in future work on query optimization, more attention should be paid to the former. An added benefit of working with R* is that we were able to get valuable feedback about the database side of our ideas from some of the project researchers. Two final (and practical) reasons for studying R* were that (1) the system was graciously made available for our experiments, and that (2) among the few DDBMS's that were in existence, R* was the one with the most powerful functionality and the greatest dependability.

4.2. Experiment Design

In this section we describe the goals of the experiments, the experimental method followed, and the hardware and software configurations used.

4.2.1. Objectives

We had two main goals in these experiments: (1) to show that load balancing can indeed make a sizable difference in the performance of database queries, and (2) to show that, in order for such improvements to be possible, the system in question need not be inordinately unbalanced. Furthermore, as a secondary goal of our study, we wanted to study the algorithms used in the system, in order to gauge the difficulty of actually implementing load balancing in real systems; it was clear to us that any load balancing schemes that could be seriously proposed as alternatives to be considered by system designers would have to be simple and straightforward to implement. We cannot expect designers of current DDBMS's to re-code a large part of their existing software, even if the reward for them were that their systems could reap substantial performance improvements. Thus, we wanted to be able to develop easily implementable load balancing strategies that would provide most of the performance improvement obtainable through load balancing (i.e., gain a lot by working a little). Also, we wanted to avoid schemes that consumed a lot of CPU cycles in order to determine the least loaded sites and make the load balancing decision. In this, we were guided by one of Butler Lampson's dictums: "In allocating resources strive to avoid disaster, rather than to attain an optimum" [Lampson1983]. Hence, another secondary goal of our study was to show that, even without trying to apply an "optimal" strategy (however that may be defined for a particular application), we could obtain sizable performance improvements.

4.2.2. Experimental Method

Our approach essentially consisted of timing the execution speed of a set of query plans for a given query under a variety of network loads. That is, first, we imposed different load levels on different sites. Then, the different plans were executed, and, of course, they performed differently as the system load changed. Under some load conditions, the plan chosen by the optimizer as the "best" did not perform optimally (thus, an optimizer that took into account the load in the system could possibly construct better-performing query plans). By this method, we were able to gauge the impact that system load can have on the choice of a query plan.

Specifically, the experimental method consisted of the following series of steps:

(1) Creating a distributed database of sufficiently large size.

(2) Developing a synthetic database workload.

(3) Coding a series of dummy programs that would artificially load the system.

(4) Modifying the cost estimates used by the R* optimizer in order to force the selection of different plans.

(5) Timing the database queries as they ran under a variety of system loads while using different query execution plans.

The following sub-sections provide further detail about each of the steps outlined above.

4.2.2.1. The Database

In the Summer of 1983, we created a distributed database in two of the computers of the Highly Available Systems Laboratory at IBM's San Jose Research Laboratory. This database was then used for a preliminary performance study of R*, which involved various members of the R* group. We decided to use this synthetic database for our new experiments because a real-world database was not available.

Two processors were used in our experiments, and they each contained identical local databases, although that was not a prerequisite for our study; the strategies we studied did not depend on the homogeneity of the local databases. Each database consisted of 8 relations, ranging in size from 10Kbytes to 14.4 Mbytes, and from 25 to 144,000 tuples. The total local database size was 45 Mbytes, of which 34 Mbytes were user data, and the rest was dedicated to secondary indices.

Further information about the relations used in the experiments described in this chapter is provided in Figure 4.1. The column labeled **indices** lists the tuple fields on which secondary indices were created. The indices on fields A1 and B1 are "clustered" indices: the order in which the tuples are physically stored on the disks corresponds approximately to the value of the tuple field on which the index is created. The indices on fields C1 and D1 are not only clustered but "unique" secondary indices: there can be at most one tuple for any given value of the index field. Relations R1 and R3 were located in one database; R2 and R4 in the other. In Figures 4.2, 4.3, 4.4, and 4.5 we show the format of the tuples in each relation.

relation name	indices	record length (bytes)	number of tuples	user data (bytes)	total size (bytes)
R1	A1, A2	100	144,000	14,400,000	19,200,000
R2	B1, B2	100	144,000	14,400,000	19,200,000
R3	C1, C2	400	18,000	7,200,000	9,600,000
R4	D1, D2	400	18,000	7,200,000	9,600,000

Figure 4.1: Relation Information

field	size (bytes)	contents
A1	10	string, sequentially assigned in range "0000000010" to "0000600000" (by 10's)
A2	10	string, randomly assigned in the same range as A1
A3	4	random integer in the range 1 to 300
...	76	7 additional fields

Figure 4.2: Relation R1

field	size (bytes)	contents
B1	10	string, sequentially assigned in range "0000000010" to "0000600000" (by 10's)
B2	10	string, randomly assigned in the same range as A1
B3	4	random integer in the range 1 to 300
...	76	7 additional fields

Figure 4.3: Relation R2

field	size (bytes)	contents
C1	10	string, sequentially assigned in range "0000000010" to "0000600000" (by 10's)
C2	2	random integer in the range 1 to 30
C3	25	a fixed string for all the tuples
C4	4	random integer in the range 0 to 29999
...	359	11 additional fields

Figure 4.4: Relation R3

field	size (bytes)	contents
D1	10	string, sequentially assigned in range "0000000010" to "0000600000" (by 10's)
D2	2	random integer in the range 1 to 30
D3	25	a fixed string for all the tuples
D4	4	random integer in the range 0 to 29999
...	359	11 additional fields

Figure 4.5: Relation R4

4.2.2.2. The Workload

In order to generate a database workload, a special tool was created. It consisted of a menu-driven PL/1 program that was able to perform any of a number of queries. The program made available to the user three general types of queries:

(1) local queries.

For these queries, all the data required to answer them is available locally (the "local" site in this context is the processor where the user is logged on). We assume that all the work done on behalf of these queries is performed at the local machine. This need not be so; for example, in a DDBMS that supports replicated

data, the process of updating a local file may require some work to be performed at the other locations where copies of the data reside.

(2) remote queries.

In these cases, all the data queried is stored at a remote site (or at a collection of remote sites). Most of the work involved in the query is done remotely; however, some local work has to be performed. For example, the local system may have to spend processor cycles sending messages to the remote site(s) to set up the query and displaying information on the user's screen.

(3) mixed queries.

These queries require both local and remote information. Different queries may have widely dissimilar ratios of local to remote data. It is clear that for these queries non-trivial amounts of work are done at all the sites containing the data. The actual amount of work done at each site depends not only on the size of the relevant data stored there, but also on the physical storage of the data (e.g., clustered versus non-clustered data), the availability of secondary indices, the details of the local query plan (e.g., which join method is used), and the actual semantics of the query (e.g., some query predicates are more selective than others), to name a few of the variables that affect the total work required.

The workload program can execute any given number of the available queries, or some random mix of them. The turn-around time of each query (or set of queries) was measured using an assembly language utility (coded by Robert Yost of IBM San Jose), since the PL/1 procedure available in the system did not have sufficient accuracy. For these experiments, query turn-around time is defined as the time that passes between the point at which the first R* utility is invoked to the point at which *all* the answer tuples are received by the workload program. We chose query turn-around time as the performance index of interest for a variety of reasons: (1) it was one of the indices we had used in the simulation experiments described in the previous chapter, (2) it is a very relevant criterion in many environments of interest (such as in local area networks of single-user workstations), and (3) its measurement was relatively straightforward. To expand further on the last point, it must be mentioned that we considered measuring system throughput as well as response time, but the construction of the appropriate experiments was too time-consuming a task (given the time for which we had the laboratory available). The reasons for this difficulty lie partly in the existence of obscure interactions among the different pieces of software that make up the operating system (e.g., how does the scheduling policy of the CICS subsystem in which R* runs affect the performance of the multiple query streams one would need in order to conduct throughput experiments?), and partly in the complexities inherent in understanding the software of R* itself (e.g., how does the fact that multiple query streams are sharing the R* buffer pool affect our results?). Although the use of turn-around time as a performance metric avoided some of the complexities mentioned above, the fact that the R*

system itself (as well as the underlying communication mechanism) was designed with throughput in mind affected our results in various ways. We will come back to this point in Section 4.3.3.

The workload tool also accepted commands for computing the mean and variance of a sequence of experimental runs. This did not prove very useful for us because, sometimes, the idiosyncrasies of the communication mechanism created obvious outliers in our data, which could not be discarded automatically by the program (this will be further explained in Section 4.3.3). The program also implemented various other commands which were not used in the particular set of experiments described in this chapter. Any database errors that occurred while executing the queries were reported to the user of the tool.

A crucial point to be made is that, in order to obtain meaningful results, the caching ability of R* had to be minimized. In other words, since R* caches locally the tuples used in the queries that it has executed in the recent past, repeating a particular query many times (which is clearly needed to obtain a valid average turn-around time) could result in copies of the needed remote-site tuples residing in local storage. Instead of flushing the buffer area after each query, we made sure that each query would require different tuples each time it was executed. For example, in the case of a query that requests the salary of an employee having a particular employee number, the workload tool would normally choose the employee number randomly (although the program could be forced to select the same one; this was very useful for debugging purposes). It could be argued that these caching effects do exist in real query streams, and that, furthermore, the designers of the system took them into account. However, since the buffer pool for any R* site is shared among all the query streams in the system, and since the size of that pool is fixed at database creation time and does not depend on the number of users in the system at any one time, we could not be entirely certain of how to account for the effects of caching on our results (it should be noted that this difficulty could have been avoided if we had been able to use throughput as our performance metric).

Finally, some comments about the suitability of using load balancing for each of the three query types described above. It is clear that, for local queries, the benefits of load balancing in many cases are very limited. Unless the processing power of a system is extremely low compared to the bandwidth of the communication mechanism that is in use, it would seldom be attractive to ship local data off-site to take advantage of excess cycles elsewhere. Even if the local CPU were terribly overloaded, the network protocols involved in shipping the tuples to a remote site would still require a large amount of local processor cycles; thus, unless many more CPU cycles are required to process the query than to move it, it would seldom pay off to do so. On the other hand, if the protocols were implemented in hardware, and the data were moved without requiring the intervention of the processor (e.g., a network interface with DMA capabilities), load balancing would become more attractive. Furthermore, there are cases where a

relatively small amount of data will be used for a large number of computations. For example, an engineering database may contain design data for a VLSI chip; that data will be processed for a long time by a variety of CAD/CAM tools, and hence, it may be desirable to ship that data to a site that has more available power. This scenario is quite appropriate for a LAN consisting of relatively low-power CAD/CAM workstations and large number-crunching processors.

Remote queries are even better suited for load balancing. Since the answer to the query will eventually have to be displayed on the user's terminal (or stored in a file) in the local system, it may be cost effective to move the data from the remote to the local site and do the computation there. This becomes especially attractive if a local cache is available. In that case, the performance of queries that access "hot spots" (i.e., frequently used data) will be substantially improved. Lastly, it may not be possible or desirable to do any computation remotely. For example, if the data is kept in a network file server [Mitchell1982], it would be necessary to transport the data to a computational site; the choice of the execution site could be based on load balancing considerations.

It is clear that load balancing is ideally suited for the last type of query described above. Since in the case of mixed queries some work is normally required at all sites, and some transfer of data must take place, it is evident that these queries present ample opportunities for taking advantage of unequal loads (and capacities) among the participating processors. In particular, there probably has to be less of an imbalance among the network processors in order for load balancing to be of use, and the relative cost of computation to communication perhaps does not have to be as favorable as in the case of the other types of queries.

In our load balancing work we were guided by the following "canonical" scenario. Consider a system with a heterogeneous collection of processing nodes. A user, who is logged on at site A, queries the DDBMS. Site A has part of the data involved in the query, and site B has the rest of the required data. Site C has none of the data locally, but has excess processing capacity. Clearly, there are three possible alternatives: we can (1) move the data at B to A, and compute there (the result is then available locally to the user); (2) perform an equivalent data move from A to B (and, after computing the answer to the query, send it to A); (3) move all the data to C, compute there, and send the final answer to A (note that the two data moves may go on in parallel in many networks). Unfortunately, we could not quite reproduce this canonical case for two reasons: first, we had only two processors available; secondly, the plans we were able to study did not perform their work in such clear-cut fashion (see the comments about merge-join plans in Section 4.3.3).

The queries studied in this chapter are of the mixed type; we will see that for R*, even under very unfavorable circumstances for load balancing (the communication channel was extremely slow and the processors were identical in their capabilities), we could still obtain sizable performance gains for this type of query by balancing the load. One

final point that applies to all three types of queries is that, in the case of multi-query transactions, it may be advantageous to transport the data to another site for a particular query if succeeding queries in the transaction would be better served by having the data at that particular site. We did not address these considerations in our work.

4.2.2.3. Generating the System Load

In order to conduct our experiments, we needed a mechanism for creating a given level of load in each of the two processors available for the experiment. We chose to maintain the required load by creating a set of compute-bound jobs that would run at the same time as our workload tool. These jobs would enter a computational loop, and then sleep for a given amount of time. By changing the ratio of the time spent computing to the time spent sleeping, we could create jobs that imposed different amounts of load in the system. Thus, by using a suitable arrangement of them, we could fine-tune the "background" CPU load that the database queries encountered in the system; this gave us an easily controllable loading method.

Note that this is a somewhat one-dimensional view of the load, since we were not similarly loading the disks. We can justify this by claiming that, in most environments of interest, the main user of the disks is the DDBMS, and that the disk utilization of the other users is a secondary effect. Furthermore, if other users have their data in different disks (and different disk channels) than those with the data used by the DDBMS, their disk accesses may not conflict with those of R* in a significant manner. However, there will possibly be conflicts with other DDBMS jobs; we have not addressed this problem, although a more comprehensive view of load (that included DDBMS disk usage) could easily be incorporated in the implementation alternatives we suggest in Section 4.4.

To ensure that the load was accurately repeatable, we had to consider the scheduling policy of the system. Although we were the only user of the two processors, there were various processes constantly running, which were needed in our experiments (e.g., the inter-processor communication mechanism). The R* DDBMS runs within a partition of the Customer Information Control System (CICS) [IBM]. This is essentially a sub-system running on top of the MVS operating system (which was the operating system used in the processors); CICS does its own process scheduling, and it supports a variety of job classes, each with different properties. We first made sure that CICS ran at a higher priority (as far as MVS scheduling was concerned) than any other activity in the system. Then we ran our loading jobs within the same CICS partition as R*. We placed loading jobs in a different priority class than R*; the properties of this CICS scheduling class were such that the priority of any job in that class was always higher than the priority of any R* job (with respect to CICS internal scheduling). Thus, no matter for how long our loading jobs executed, they would always have a higher priority than R*.

For the experiments presented in this chapter, we defined load as the percentage of the CPU that was utilized. We understand that this may not be a complete (or even a

very good) estimator of load. However, it had the advantage of being easily available (we used the features of the RMF monitor available for MVS systems). We were unable to measure the equivalent of UNIX's load average described in Chapter 2. This would have been preferable since it would have enabled us to use a consistent load metric for all the work described in this thesis. We did not choose to use the CPU utilization for our INGRES work because of some uncertainties about the accuracy of the measurement of that index in UNIX [Hagmann1983]. On the other hand, one advantage of using two different load metrics in our work is that the success of both experiments (the R* and the distributed INGRES) shows that implementing load balancing with either of two "reasonable" load metrics leads to performance improvements over the non-load balancing implementations. We cannot say much more about the proper choice of load metric for any particular DDBMS; that issue remains the topic of future research.

4.2.2.4. Modifying the Optimizer Plans

In order to test the effect of the system's load on the performance of queries, the optimizer had to be forced to pick different plans for the same query (normally only one "best" plan is chosen, in the manner described in a previous section). Thus, the different plans were identical *semantically* (i.e., the answer to the query should be the same for all plans) but not *procedurally* (i.e., the actions of the database system were quite different in all the plans).

First, we modified the PL/1 workload program so that each query to be studied was included twice in the program. Then we compiled the program; by this we do not mean language compilation, but rather database query compilation (see [Daniels1982] for more details). We interrupted the query compilation process just before the optimizer chose a plan, but after it had gathered all the estimates needed for compilation (as was mentioned before, those estimates are related to CPU utilization, number of I/O's, and number of network messages required). The interruption was effected through the use of a high level debugger created by the R* implementors for the purposes of system development. After the appropriate breakpoints were reached, the time estimates that the optimizer had obtained were modified in such a way as to make some particular plan seem the most attractive. Thus, we were able to force it to pick the different plans required for the experiment.

It should be pointed out that we were not able to force the compiler to pick an arbitrary plan. Due to the complexity of the data structures used in the system, we restricted our choice to the plans that were the final candidates for the "best" plan. Specifically, for each site involved in the query (and for an extra "unknown" site), the compiler first determines which plan would be the best if the final answer had to be delivered to that particular site. It then chooses the actual query plan by introducing into its cost model the expense of moving the answer set to the user's site. We stopped the planner just before it selected this final plan (i.e., we could only experiment with the plans that "reached the semifinals"). Although we lost a great deal of flexibility because

of this, the plans available proved sufficiently varied for our purposes.

Understanding the intricacies of the compiler sufficiently well to modify its plans proved a major hurdle in our work. This was not so time-consuming because of the intricacies of the actual details of the modification process, but was rather due to the author's lack of familiarity with the system.

4.2.2.5. Running the Queries

As mentioned before, the experiments were performed in stand-alone fashion. Each of the queries that were used in the experiments required a large amount of real time to study (i.e., to set up the systems, modify the plans, execute the query multiple times, and so on). In particular, the queries described in this chapter each required about a week of experimenter time to study. This was due to a variety of reasons: (1) since the queries were sometimes executed under high system load, the actual turn-around time could be large; (2) for each point in the load space, the competing plans had to be executed a number of times to achieve statistically significant results; (3) many points in the load space were considered; and (4) the system itself was only available for experimentation at certain times.

The laboratory setup was available for only a limited period of time, due to an impending change in the hardware installed. This, combined with the long experiment times, precluded our experimenting with a large number of queries, but we managed to complete about 15 experiments. In Section 4.3 we discuss two of the experiments carried out, and we explain why load balancing was not appropriate for some queries.

4.2.3. System Configuration

As previously mentioned, the experiments were carried out on the equipment of the Highly Available Systems Laboratory. We were able to secure the use of only two of the processors in the laboratory. Each of the two was identical in hardware and software configuration. The processors were IBM 4341-M2 models (about 1.3 MIPS) with 8 Mbytes of main memory each. The data was kept on IBM 3350 disks, which have a mean access time of 25 ms and a data rate of about 1.2 Mbytes per second. Both machines were running IBM's MVS operating system.

The processors were linked by both a channel-to-channel connector and by an IBM 3705 communication device. Due to hardware problems with the 3705 we were able to use only the channel-to-channel link in the experiments. Using this link did not improve the communication time in a significant way because of the large delays imposed by the communication software. R* executes as a CICS resource manager, and the communication mechanism that CICS uses is IBM's Virtual Telecommunications Access Method (VTAM). According to published data [Selinger1980], a reasonable estimate is that the system requires about 40,000 instructions to send and acknowledge a message. On a 1.3 MIPS systems that translates to roughly 31 milliseconds of software overhead. That should be compared with less than 3 milliseconds for a send-receive-reply message round

trip in other networking schemes [Cheriton1983].

4.3. Results

In this section we describe some of the results of our experiments. The first two sub-sections show how, in two of the successful experiments, we were able to obtain marked improvements through load balancing. The third sub-section explains why, for some queries, load balancing experiments either could not be performed (due to faulty software) or there was no possible improvement for the plans (because of the nature of the communication channel).

4.3.1. R* Experiment #1

The query studied in the first experiment was the following:

```
SELECT  R1.A1, R1.A2, R2.B1, R2.B2
FROM  R1, R2
WHERE  R1.A3 = R2.B3 AND R1.A1 = RANDOM
      AND R2.B1 = RANDOM;
```

R1 and R2, the relations involved, were described in Section 4.2. In the experiment, R1 was a local relation, and R2 was at the remote site. RANDOM stands for a variable whose value is assigned randomly at run time (**not** at compile-time); the value of RANDOM is chosen (with a uniform distribution) from the entire range of possible values for A1 (which is the same as B1). Informally, the query is asking the DDBMS for the tuples in relations R1 and R2 that have common values in two fields (1 and 3). The execution of this query consists of forming two temporary relations by selecting tuples from relation R1 (those with RANDOM in field A1) and similarly selecting tuples from R2; then joining the resulting relations on field A3 and B3 to form another temporary relation; finally, obtaining the answer set for the query by projecting fields A1, A2, B1, and B2 from that relation. Note that a semantically identical query could be expressed as follows:

```
SELECT  R1.A1, R1.A2, R2.B1, R2.B2
FROM  R1, R2
WHERE R1.A1 = RANDOM AND R1.A3 = R2.B3 AND R1.A1 = R2.B1;
```

However, the actual sequence of actions followed by the planner in this second case left us with a different set of "semifinal" candidates for our experiments (although the compiler still chose the same "winner"); the candidates were unsuitable because they consisted of merge-join plans (see Section 4.3.3 for more details on this).

We chose two plans for the experiment. Plan A required that the DDBMS perform most of the computation locally, while in plan B most of the work was to be done at the remote site. Specifically, both plans called for an index scan on the outer relation of a nested-loop join; the inner relation of the join was R1 (the local one) in plan A, and R2 (the remote relation) in plan B. Query execution would be terminated by a final predicate selection (at the site of the inner relation). Less formally, in plan A, the system

was to scan (using the secondary index on field B1) relation R2, picking out the tuples obeying the predicate "R2.B1=RANDOM". Those tuples would then be transferred to the local site (not individually, but as many as could be fitted in a message). Then the DDBMS would perform a local join by scanning the tuples sent over once for each tuple in R1; those whose B3 field equaled the A3 field of a local tuple were chosen. Finally, the joined tuples would be further winnowed by discarding those whose A1 fields were not equal to RANDOM (note that, for this query, since the same value of RANDOM was used in two places in the query and the relations are identical, this final process does not discard any tuples). Plan B was similar, but with the remote relation (R2) playing the part that R1 performed in plan A.

Note that plan B has to do intrinsically more work than plan A. This is so because, after the query answer set is determined, the tuples have to be shipped to the user site in plan B, while they are already in the correct place for plan A (for this particular query, the actual extra overhead involved in transferring the answer back to the user's site was one extra message). Since the optimizer is aware of this, it normally would select plan A.

However, if the remote node were more powerful than the local one, and if the optimizer had information on the processing speeds of different nodes, it might assign different costs to the computational part of the two plans, and might then conceivably pick plan B as the most effective one. Although currently the optimizer does not have cost information on a per site basis (it uses the same processing speed and message delay estimates for all nodes), the information will have to be added to it sometime in the future for obvious reasons. For example, if R* is ever to run in an environment composed of workstations and mainframes, some measure of CPU performance will have to be incorporated into the optimizer. Heterogeneous CPU's are not the only reason for having processor-specific costs; in a long haul network, the cost of sending messages is dependent on the distance to the destination site. Hence, ultimately the optimizer will have to be revised so that it can keep (and take into account) all appropriate information about each site with which it intends to interact. If the number of processors on the network becomes extremely large, or there is not much local storage, that information could be stored at some subset of the nodes and cached locally at each site where queries are compiled.

Actually, this caching issue must eventually be addressed, since a similar problem occurs with other database directory information. For example, in a very large network, keeping track of the site(s) at which all the relations reside might require inordinately large amounts of storage. In such networks, perhaps the best place for this type of information would be a name server [Oppen1983, Birrell1982], a mechanism that is required in large distributed applications.

In Figure 4.6 we present in graphical form the results of the first experiment. On the x axis we report system load (i.e., CPU utilization) at the local site. Similarly, the position along the y axis represents the load on the remote processor. For points in this

load space above the shaded area (i.e., mostly points with x-coordinates less than their y-coordinates), plan A had better performance (i.e., lower turn-around time). For points below the shaded area, plan B offered performance improvements of various magnitudes. The shaded area comprises points for which the performances of both plans were about the same; we defined this to mean that the means of the turn-around times of plans A and B differed by less than the sum of the respective standard deviations. (Note: the boundaries of the shaded area are drawn so that they approximately enclose the points where both plans performed about equally). In order to gauge the actual size of the improvement possible by using plan B, we have further marked three points on Figure 4.6, and labeled them with the performance degradations that using plan A would entail when compared to using plan B at those points.

In Figure 4.7 we present in tabular form the raw results obtained in the experiment. All turn-around times in the table are normalized so that plan A, with no background load in either system, takes one unit of time. Both plans were executed from five to ten times for each of the background load combinations listed in the table (as previously mentioned, load is defined in these experiments as percentage of CPU utilization). The table also gives the standard deviation of the measurements. Degradation is defined as the difference between the turn-around time of the plan normally chosen by the optimizer and that of the alternate plan, normalized by dividing this difference by the time of the alternate plan. Negative degradations indicate that the "normal" plan performs better at that point. Obviously, not every point in the space was tried, but that was not necessary, since the shape of the shaded region contains all the relevant information for deciding between the two plans.

As can be seen from the table in Figure 4.7, large performance improvements (up to almost 30%) are possible if the optimizer takes the loads existing on the machines into account. The graph in Figure 4.6 shows that under many circumstances (about half the time for this particular query) the optimizer chooses the wrong plan; furthermore, the loads of the two systems do not have to be greatly unbalanced before the differences become appreciable. Note that we did not even drive either of the two nodes in the system to saturation, which could indeed be the case in many a real installation. Greater load imbalances would strengthen our results; so would the availability of a more heterogeneous set of processors, since a remote host whose power were much greater than the local host would be an ideal target for offloading local tasks.

Some final observations about the diagram in Figure 4.6 are in order. The spread of the shaded region is related to the variance present in the experimental setup. Decreasing various sources of variance in the experiment (for example, if the response time of the message passing mechanism had lower variance) would result in a decrease of the shaded region's size. If one of the two machines were far more powerful (e.g., higher MIPS rate or faster disk access time) than the other, the shaded region would move closer to the axis labeled with the load of the more powerful hardware; this is so because, if that were the case, the load would have to become very unbalanced (i.e., the

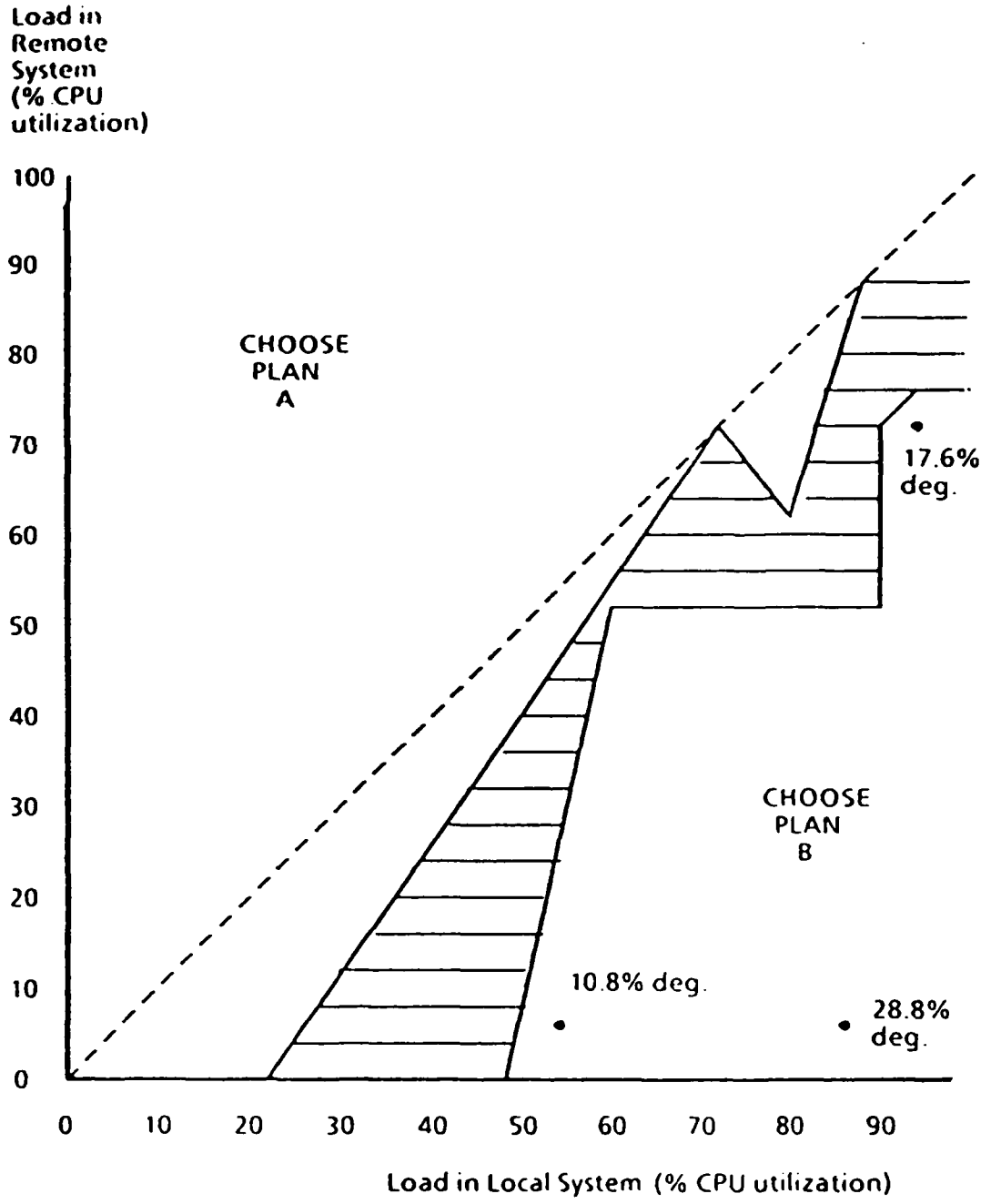


Figure 4.6: Results of Experiment #1

background load		turn-around time				
		Plan A		Plan B		degradation
local	remote	mean	st. dev.	mean	st. dev.	(%)
5	5	1.00	0.06	1.02	0.04	-2.0
42	5	1.20	0.12	1.25	0.06	-4.0
57	5	1.62	0.07	1.47	0.11	10.2
87	5	3.93	0.17	3.05	0.25	28.8
57	42	1.61	0.12	1.72	0.16	-6.4
67	42	2.28	0.17	2.03	0.04	12.3
67	57	2.35	0.16	2.29	0.30	2.6
72	57	2.67	0.31	2.38	0.12	12.2
80	57	3.27	0.33	3.06	0.04	6.9
87	57	4.12	0.38	3.66	0.24	12.6
94	57	6.90	0.77	6.54	0.78	5.5
72	67	3.01	0.21	2.96	0.17	1.7
80	67	3.38	0.11	3.36	0.28	0.6
87	67	4.53	0.33	4.23	0.13	7.0
94	67	8.54	0.67	6.94	0.36	23.0
87	72	4.59	0.52	4.52	0.17	1.5
94	72	8.02	0.24	6.82	0.34	17.6
87	80	6.15	0.37	5.55	0.81	10.8
94	80	8.31	0.45	8.02	0.49	3.6

Figure 4.7: Results of Exp. #1 - Tabular Form

more powerful machine severely overloaded and the other processor almost idle) before it would be preferable to execute the plan that carries out most of its work at the less powerful machine. (In this case, as was pointed out in the previous paragraph, if the more powerful processor were the remote one, load balancing optimizers could obtain sizable performance improvements by executing local queries remotely.) For this particular query, the same amount of work is carried out under both plans, except for an extra message in plan B. Thus, if one could send messages at a faster rate, the cost of this extra work would be minimized, which would result in the shaded region shifting closer to the dashed line drawn at a 45 degree angle. Note that, as both sites become more loaded and response time goes up, the overhead of sending messages becomes negligible. This is shown by the fact that the shaded region's edges get closer to the 45 degree line in high-load cases.

4.3.2. R* Experiment #2

The query used for the second experiment was as follows:

```
SELECT  R3.C1, R3.C2, R4.D1, R4.D2
FROM  R3, R4
WHERE  R3.C1 = R4.D1
   AND R3.C2 = RANDOM
      AND R4.D2 = RANDOM
      AND R3.C4 > VAR1
      AND R4.D4 > VAR2;
```

Now the relations involved are R3 and R4 (see Section 4.2 for their description). R3 was the local relation and R4 the remote one in this experiment. RANDOM again stands for a variable whose value is assigned randomly at runtime (again, with a uniform distribution drawn from the entire range of possible values for fields C2 and D2). VAR1 and VAR2 are two (different) variables, whose value is set at runtime. It is important to note that this implies that the optimizer can only make a wild guess at what their value will actually be at runtime. As will be seen later, this means that the DDBMS will not be able to predict with any certainty which plan will be optimal for these queries.

The semantics of this query are somewhat similar to those of the query used in experiment #1, with the exception that now we are further restricting the answer set tuples by imposing two new predicates: "R3.C4 > VAR1" and "R4.D4 > VAR2". As in the first experiment, the syntax of the query was chosen so that a certain set of plans could be generated.

For this experiment, we again chose to study two different plans (C and D). The description of plan C is identical to that of plan A in the previous section, with the difference that, as the index scan is being done on the outer relation, the tuples that do not match the predicate "R4.D4 > VAR2" are discarded, and, after the join, the tuples not matching "R3.C4 > VAR1" are also dropped. A similar relationship holds between plan D and the previously described plan B. Essentially, in plan C we will move the relevant tuples of R4 to the local machine and perform the join locally, whereas in plan D we move the tuples of the local relation (R3) to the remote site.

Note that the symmetry existing in the previous experiment is missing here, since the amount of data involved in the query is different at each of the two sites (as it depends on the respective selectivities of the "greater than" predicates). In other words, the run-time values of VAR1 and VAR2 determine the number of tuples selected at both sites. Hence, we cannot say *a priori* which plan has intrinsically to perform more work. However, we will see that the relative loadings of the processors still matter. One final piece of information is that, under normal circumstances, the optimizer would always select plan C, in spite of the fact that plan D might be better even in the absence of load. In this section we will describe the experimental results obtained using

two different sets of values for VAR1 and VAR2.

In Figure 4.8 the results of the second experiment are shown for VAR1=15000, and VAR2=25000. This results in a query that involves about 1/2 of the tuples of the local relation, and about 1/6 of the remote tuples. That is, there is three times as much query data locally as remotely. Hence, we would expect that a load imbalance larger than in experiment #1 would be needed before load balancing pays off. As can be seen in Figure 4.8 (the x and y axes are labeled as before), even in this biased case load balancing pays off in a non-trivial portion of the cases, and the potential gains are not negligible (see the table in Figure 4.9).

In Figure 4.9 we present in tabular form the raw results obtained in the experiment. We again normalize all turn-around times in the table so that plan C, with no background load in either system, takes one unit of time.

In Figure 4.10 we show the results of the same experiment, but for VAR1=25000 and VAR2=15000, so that now most of the data is at the remote processor. While the optimizer would have still selected plan C for this query, it is clear from the graph that plan D should have been chosen in most cases. This is so because now it is better to move the relatively smaller amount of local data to the remote machine. Our results again show that this decision is very much dependent on the load, since plan D is a better choice than plan C for most of the load space. The raw data for this experimental run is presented in Figure 4.11.

We have again shown that the load conditions existing in a distributed system need to be taken into account by the optimizer. But there is another point to be made here. The optimizer still made an incorrect choice of plans even neglecting the loads, since crucial runtime information (i.e., the values of some of the query variables) was not available to it at compile time. We feel that future query optimizers should be built in a way that allows the system to reconsider planning decisions made at compile time in the face of new information about either the state of the network or the query itself. We will have more to say about this runtime vs. compile-time tradeoff in the concluding section of this chapter. The implementation implications of doing all this extra work at runtime (at least for the case of load balancing) are considered Section 4.4.

4.3.3. Experimental Problems

We found many queries (about a third of those we studied) for which we could not carry out successful load balancing experiments. This happened for a variety of reasons, seldom because load balancing did not offer substantial benefits, but, most often, because of experimental problems. For example, in a few cases, there was so much more local data than remote, that it never paid off to move the query off-site unless there was an inordinate imbalance in the loads. (Although it should be pointed out that employing a load balancing optimizer would not be detrimental in these cases, since that balancer would simply choose to run locally.) In other experiments, the candidate plans

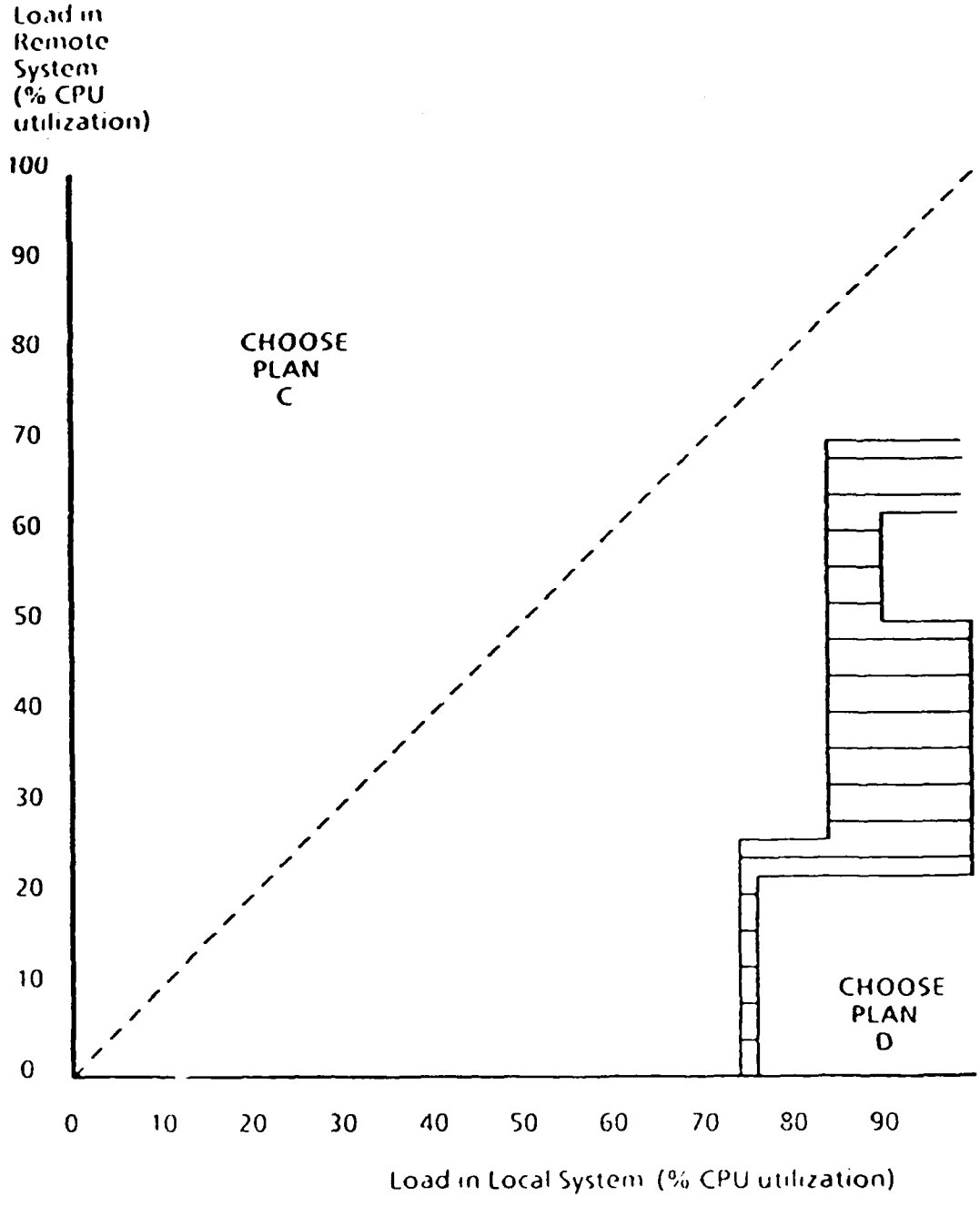


Figure 4.8: Results of Experiment #2, Case I

background load		turn-around time				
		Plan C		Plan D		degradation
local	remote	mean	st. dev.	mean	st. dev.	(%)
5	5	1.00	0.07	1.26	0.04	-20.6
67	5	2.07	0.17	2.25	0.09	-8.0
72	5	2.47	0.15	2.43	0.04	1.6
80	5	3.24	0.11	3.29	0.10	-1.5
87	5	4.86	0.24	4.28	0.24	13.6
94	5	8.84	0.28	7.19	0.41	22.9
72	42	2.62	0.12	2.88	0.16	-9.0
80	42	3.43	0.18	3.62	0.14	-5.2
87	42	4.86	0.36	4.71	0.16	3.2
94	42	8.73	0.33	8.16	0.84	7.0
87	56	5.16	0.23	5.41	0.46	-4.6
94	56	9.09	0.43	8.19	0.44	11.0
87	67	5.26	0.24	5.59	0.20	-5.9
94	67	10.07	1.22	9.18	0.71	9.7
94	80	10.20	0.50	10.62	0.80	-4.0

Figure 4.9: Results of Exp. #2, Case I - Tabular Form

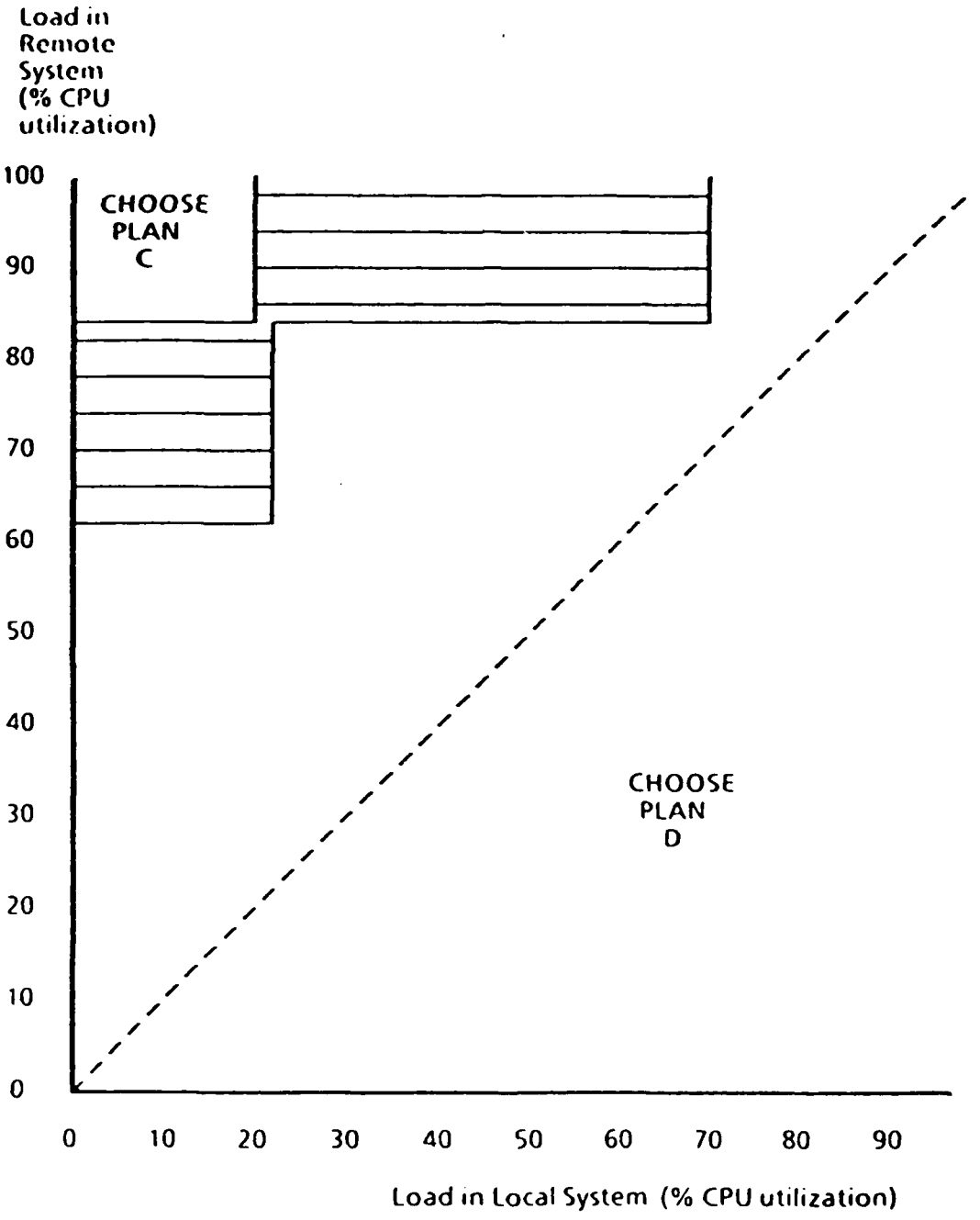


Figure 4.10: Results of Experiment #2, Case II

background load		turn-around time				
		Plan C		Plan D		degradation
local	remote	mean	st. dev.	mean	st. dev.	(%)
5	5	1.00	0.03	0.78	0.04	28.2
5	67	1.68	0.14	1.58	0.05	6.3
5	72	1.99	0.11	1.83	0.07	8.7
5	80	2.33	0.12	2.47	0.09	-5.7
5	87	3.29	0.23	3.59	0.06	-8.4
42	80	2.64	0.14	2.43	0.05	8.6
42	87	3.68	0.27	3.69	0.20	-0.3
42	94	5.85	0.39	6.43	0.41	-9.0
56	87	3.94	0.20	3.71	0.18	6.2
67	87	4.06	0.10	3.94	0.15	3.0
80	87	4.91	0.18	4.10	0.25	19.8

Figure 4.11: Results of Exp. #2, Case II - Tabular Form

that remained as "finalists" in the optimization process (as described previously) were such that there was only one "good" plan and no reasonable alternative. We tried to modify the system so that plans discarded in previous rounds would not be pruned, but this proved too complex to accomplish in our limited time frame. For still other queries, while load balancing offered improvements, the queries themselves were so complex that their actual execution time was too long, thereby precluding their use in our time-constrained experiments.

A major stumbling block in our work was the lack of merge-join plans. While the planner does consider merge-joins as candidates (indeed, it chooses merge-join plans much of the time), the software that implements merge-joins was inoperative during our experiments; this is to be expected in a prototype system such as R*. However, merge-joins are the ideal candidates for load balancing, since their behavior is such that they move all the appropriate data to one site and compute there (this is consistent with the "canonical" scenario introduced in Section 4.2.2.2). Clearly, one could choose the computation site according to load balancing considerations. Merge-joins would have provided us with a cleaner experimental setup, since the nested-loop join implementation in R* sometimes tries simultaneously to send data while it is reading some more from the disk (which is a departure from our canonical scenario - this was not the only time that we had to reconsider some of the assumptions we made before the experiments).

Another major problem was that the message passing software was so slow (relative to the computational requirements of the queries) that, in many cases, what would have been an attractive alternative plan (in a system with less communication overhead) was slower just because it sent a few more messages than the plan chosen by the optimizer

(in our experiments, a query's total processor residence time was approximately equal to the time required to send a few messages; thus, if message sending were faster or queries required more processing cycles, this problem would have been less important). The communication mechanism also had the annoying habit of sometimes holding onto data that should have been sent, while it waited for more send requests. It took many hours of careful analysis of system traces to determine that the culprit for the obvious outliers in our experiments was the communication sub-system. Even though many experts were consulted about this problem, we were unable to find a cure.

It should also be explained why we focused on joins in our experiments. This was so for a variety of reasons. In the first place, the speed of communications was such that we had to look for queries that performed a significant amount of work in order for our experiments to be meaningful. Secondly, we felt that, since joins are the most time-consuming operations in a relational DDBMS, it made sense to focus on them rather than on other operations. Lastly, we did not look at groups of queries (i.e., complex transactions) because the planner optimizes one SQL statement at a time. Changing its algorithms to optimize for a group of queries would have taken far more time than we had available for the experiments.

But the main difficulties of the experiments lied in the complexity of the software involved and the obscure interactions in the system. Our lack of familiarity with both R* and the rest of the system software (MVS, CICS, VTAM, and so on) was an aggravating factor. Learning how the optimizer worked, and how to modify its cost estimates took an inordinately long amount of time. Using turnaround time as a performance metric in a system that had been designed for throughput created unexpected problems in a variety of places (for example, if we had been using throughput instead, the problem mentioned above of the communication software holding onto data for a while would have had lesser importance). Finally, the sheer amount of real time required by each experiment limited the number of queries we could explore.

4.4. Implementation Considerations

Although we did not actually implement load balancing strategies in the R* optimizer, we developed two alternative strategies for doing so. In this section we describe them and briefly consider their performance and their cost.

Both suggested implementations require that the system somehow detect an abnormal load level in some of the sites involved in the execution of a query. As was pointed out in a previous section, performing some runtime checking is a necessity in R*. This is so because there may have been changes in the system (e.g., an index being deleted) since the time the query was compiled, that invalidate the plan chosen by the optimizer (e.g., if the plan used the deleted index). Thus, there would not be as much overhead involved in detecting abnormal load conditions and invalidating the plan as there would be if no runtime checks had already to be made. There should not be any extra work involved in computing the load metric (the CPU utilization in this case), since the

underlying operating system normally maintains that information, unless we found that this information was not sampled or smoothed properly. The information about the loads does not have to be broadcast, as in some of the load balancing schemes described in previous chapters, because the user site must contact all relevant remote sites before executing the query (in order to ensure that the sites involved are still functional and that the data and indices used by the query plan are still in place).

At this point, there are two possible alternatives. The first design requires the DDBMS to re-compile the query under abnormal load distributions (for the initial compilation, the optimizer may assume whatever "normal" load distribution it expects to find). This has the disadvantage of requiring the extra work of a new compilation. Thus, the imbalance has to be sufficiently large to warrant the effort. To make matters worse, this extra work may take place while there is a high load in the system (although one appealing approach is to try to carry out as much of the compilation work as possible on the least loaded processors). The main advantage of this alternative is that it is simple to implement (the system is already able to re-compile at any time), and does not require the extra storage that the second scheme demands.

The second approach forces the optimizer to compute more than a single plan for each query. The planner would keep a few different plans, each of which would be appropriate for different load conditions. This would involve a couple of minor modifications to the optimizer: (1) changing it so that it does not eliminate all plans except one, and (2) varying the costs it considers, so that load is taken into account (this change is actually needed also in the first approach). The main advantage of this method is that practically no further major amount of work is needed at runtime due to changing load conditions. The drawback is that compilation is a little slower and there are extra storage requirements. Although it is hard to estimate the extra storage needed, rough calculations show that we would require about 600 bytes per SQL statement (per plan kept).

It should be pointed out that modifying the optimizer to consider load in its planning is relatively easy to implement. As we mentioned during our previous discussion of processor-specific costs, the optimizer will eventually need to be modified so as to compute different processing costs for different hardware configurations. Once these changes are implemented in the compiler, variations in a processor's load can be modeled by changing the processing costs associated with that node.

Thus, we have presented two ways to implement load balancing in DDBMS's. Both approaches are expected to be simple to implement and not very costly, in either compile or running time. Those R* research group members with whom we discussed our approaches could see no obvious fault with them.

One final point is that, in some environments, there are a few, frequently used queries, or a few very time-consuming queries. In those cases, it may be possible for the users to employ the same approach we used in our experiments, and determine

empirically what are the appropriate plans for different sections of the load space. The users could then (manually) run the correct plan for the conditions that exist when they want to execute the queries. Of course, this is the approach of last resort, and certainly not as effective as the alternatives described above. A refinement of this alternative would be to automatically trigger load balancing only for these kinds of queries.

4.5. Conclusions

In this chapter we have presented our R* experiments. The results have shown that load balancing strategies offer the possibility of sizable improvements in DDBMS performance under reasonable load conditions. Furthermore, these strategies are neither complex to implement (we have suggested two simple implementations) nor expensive to run (at either compilation or execution time). The load metric used was a simple one, and seemed to be sufficient for a real implementation. With more heterogeneity in the hardware, we would expect load balancing strategies to be even more successful. Thus, we have met our experimental goals of showing load balancing to be both useful and easy to implement.

Not all the queries that we studied benefited from load balancing. The reasons for this were, for the most part, related to the low speed of the communication software and to the temporary absence of merge-joins.

We do not feel that adding load balancing to a sophisticated optimizer such as R*'s is "polishing a round ball". We have shown that runtime considerations (which have been somewhat neglected in previous work) are very important to the performance of a query planner. Some researchers have focused on obtaining finer grained statistical information about the distribution of tuple values in the database in order to improve query performance via better query compilation. Since there is so much to be gained by looking at runtime information, we feel that future work in this area should study runtime issues as well as compile time ones. This will be especially useful in LAN environments with a high degree of heterogeneity and dynamically changing conditions.

This experimental work proved to be substantially more difficult than a comparable simulation study. However, we feel that the results obtained warranted the effort, and that they offer further support for the performance improvements suggested by the findings of the preliminary simulations described in Chapter 3. An additional advantage of these experiments over simulations is that, at many times during our work, we were forced to reconsider some of our assumptions. In simulation work it is sometimes easy to build one's biases into the model or skip crucial details. It is harder to make those mistakes when one is working with a real system.

In the next chapter we will present a third study of the problem of applying load balancing strategies to DDBMS's.

CHAPTER 5

Distributed INGRES

In order to extend our work to a second distributed DBMS, we studied the effect of load balancing on the fragment-replicate query optimization strategy proposed for distributed INGRES. In this chapter, we describe the model used in that work, and present the results of the simulation experiments performed.

5.1. Introduction

As we explained in a previous chapter, it was desirable to study the applicability of load balancing to the optimizer of a second DDBMS. Distributed INGRES was chosen because the design of its query processing strategies has been well documented in the literature, and it was easily available to us. It would have been preferable to perform the experiments described in this chapter on a running version of that system. However, at the time this work was to be carried out, the software in question was not yet reliable enough to permit wide-scale testing and experimentation.

Query processing in distributed INGRES is relatively complex, and quite different from the mechanism described for R* in a previous chapter. The details of the procedure will not be completely described here; a careful description is given in [Epstein1978]. There, a ten-step optimization algorithm is presented, which deals with complex queries that access a number of (possibly) fragmented relations distributed among a multitude of sites. Basically, after sub-queries are handled, the original query is decomposed by a reduction algorithm (see [Wong1976]) into a set of irreducible queries. Then, the sites that will process the queries are chosen. Next, the algorithm decides which relation will remain fragmented, and the rest of the relevant relations are replicated at every processing site (i.e., a copy of each fragment of each relation is sent to all sites involved in the query that did not originally contain it, except, of course, for the pieces of the "fragmented" relation). At this point, only local query processing operations remain to be done; the "master" site (i.e., the site at which the query originated) broadcasts the work to be done to all the participating processors (the "slave" sites).

In [Epstein1978] two optimality criteria for query processing policies are considered: one consists of minimizing transmission costs, the other execution time. Reasonable heuristics are suggested for the implementation of both types of policies. In order to decrease transmission costs, the heuristic proposed is to select as the fragmented relation the one with the largest amount of data, and to replicate the other relations at every site that has a piece of the fragmented relation. Actually, this is the heuristic suggested

for broadcast networks; there is a different heuristic for the case of point-to-point networks (the authors suggest moving all the data to one site in the latter case). We will henceforth refer to the broadcast network scheme as FR (for Fragment-Replicate). A further simplification of this policy would be to select at random the relation to be fragmented; this is the scheme currently implemented on distributed INGRES. We shall refer to it as RS (for Random Selection of fragmented relation).

We suggest in this chapter two ways to incorporate load balancing into the basic fragment-replicate policy. Our first proposal involves selecting the relation to be fragmented on the basis of a weighted sum of the local sizes of the data. Specifically, in a network with N sites, the relation chosen to remain fragmented is the one that maximizes

$$\sum_{i=1}^{N}(d_i/l_i)$$

where d_i is the size of the relation fragment at the i-th site, and l_i is the load at the i-th site. Actually, l_i will be the value of some load metric at the i-th site; we will assume that any load metric of interest can be defined so that a higher value of the metric implies a larger level of load, and that its value is always positive. We will refer to this policy as the LB1 scheme (for Load Balancing policy #1).

The second policy incorporating load balancing that we will study in this chapter is a further refinement of LB1; we will refer to this new policy as LB2 (for Load Balancing policy #2). In all the policies discussed so far, when a join is to be computed, work is done only at the sites that have some of the data required for the join. In LB2, we attempt to make use of the idle processors that would not be involved in the join under the other strategies. We do this by acting as if we had one more relation involved in the query, one with a size equal to the size of the smallest of the relations really involved in the query, and located at the processor with the smallest load (as defined by our load metric). Then, we choose the fragmented relation in exactly the same manner as in LB1. The effect of this strategy is that the DDBMS will act as it does under LB1 while the load in the system is relatively even, while, under load imbalance conditions, the DDBMS will copy all the data pertinent to the join to the least loaded processor, and proceed to execute there, even if that processor did not originally contain any of the data involved in the query.

At this point, we should also describe the heuristic given in[Epstein1978] to reduce the query processing time. This scheme involves "equalizing" the amount of data to be processed at each site. That is, the relation to be fragmented will be evenly distributed across all the nodes of the network (and, since the processing cost in the model used by the authors is independent of the relation chosen, the relation to be fragmented can be selected to be the one that we can evenly distribute with the least communication cost). However, if communication costs are not zero, we can force this scheme to perform arbitrarily badly by simply increasing the number of nodes in the network. Moreover, as

the authors of the paper point out, this is suitable only for an environment where all the processors have identical computational speeds. Since we are interested in policies that are applicable to heterogeneous environments, we will not consider this scheme further in our work.

In this chapter, we will present the results of simulations that compare the performance of the four query processing schemes described above: FR, RS, LB1 and LB2. The next section of this chapter describes the model used for the experiments. The third section details the experiments performed and compares the effects of the four policies on various performance criteria. The final section draws some conclusions about the desirability of using load balancing techniques in conjunction with fragment-replicate query processing strategies.

5.2. The Model

In the experiments, each host was simulated using the model shown in Figure 5.1. Although this model is similar to the one shown in Figure 3.1, the actual details of the simulation are different, in that this model attempts to mimic the behavior of a specific DDBMS (INGRES), instead of a general DDBMS, and it is a closed model (while the model described in Chapter 3 was open). The DDBMS was modeled as a collection of such processors connected by a network (the network was modeled by an exponential server).

During the simulation, there were a fixed number of users at each processor submitting database queries. Each query starts at a terminal, then goes to the CPU queue, where the DDBMS decides how to process the query (according to the strategy being simulated). When the DDBMS has chosen the fragmented relation for the join, a message is broadcast by the master site (i.e., the one where the query originated) to all the sites containing some of the join data (in the case of LB2, there may be an extra message recipient if the query will be executed at the least-loaded site as described in the previous section). The master site then waits (in the wait queue) for the other sites to complete their work. (In Figure 5.1 the triangular figures represent the coupling of two actions; the triangle with a left-pointing apex represents the two actions of sending the broadcast message and waiting for replies from the other sites; the triangle with a right-pointing apex represents the fact that the master site job will depart from the wait queue when the appropriate acknowledgments are received). When a site that contains fragments of the replicated relations receives this initial message, it will access the appropriate data on its disks, and broadcast it to all the sites containing the fragmented data. If a site that contains fragments of the replicated relations does not have any data from the fragmented relation, it is now finished with its work and it sends an acknowledgement to the master site. When fragmented data sites receive the initial message, they also obtain the relevant data from their disks, but then wait for the data from the replicated relation sites. When all the replicated data has arrived, these sites proceed to execute the local portion of the join (if there is more than one job in the

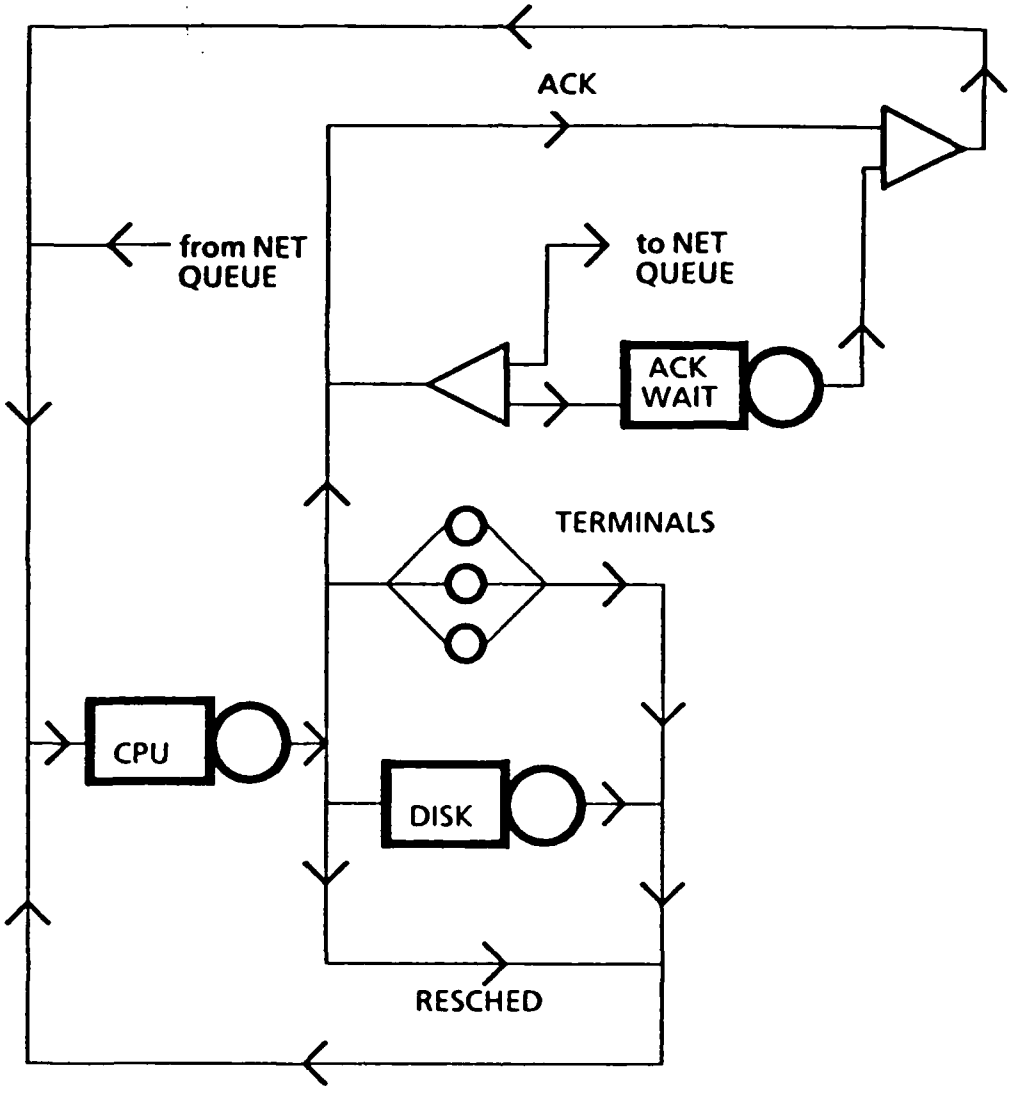


Figure 5.1: Host Model

CPU queue, each job will execute for a time-slice before being re-scheduled). We assumed that it would take about 10,000 instructions to process each page of the data involved in the local join, and that each machine can process one million instructions per second. When each local join is finished, an acknowledgment is sent to the master site. When the master site has received an acknowledgment from all the participating sites, all the work required for the query is finished, and the job goes back to the terminal, from where after some time the user submits a new query.

The queries modeled were all two-relation joins, and each relation could be composed of multiple fragments of varying sizes (although, in the simulations described in the next section, all relations consisted of single fragments). The relations involved in the queries are located throughout the network, but with a higher probability of being local (i.e., in the processor where the query originates); for example, a local data probability of 0.5 implies that, for a two-relation join, .25 of the queries are strictly local (i.e., all local data), .25 are strictly remote queries, and the rest are mixed.

The parameters used in the model are shown in Figure 5.2, together with their values. The values of the parameters were set based on the measurements we presented in Chapter 3.

We terminated each of the simulation runs when a given number of jobs had been completed. Simulations were executed for a variety of values of this limit, and the response time and throughput measures were found to be stable (within 5% of each other) for runs in which more than 100 jobs were allowed to finish. We ran the simulations whose results are reported in this chapter until at least 150-200 jobs were completed (and in some cases for longer than that).

5.3. Experimental Results

The goal of these experiments was to gauge the benefits of introducing load balancing into the fragment-replicate query processing strategies used in (or proposed for) distributed INGRES. As in previous chapters, the performance indices of interest were mean job turn-around time and system throughput. As before, mean job turn-around is defined as the mean time elapsed between job initiation and job completion, and system throughput is defined as the total number of jobs completed per second by the system.

The simulations were performed for a variety of user distributions, and for all the query processing strategies of interest (i.e., FR, RS, LB1, and LB2). The load metric used in the simulations was defined to be the number of jobs waiting in the CPU queue plus 1, divided by the mean CPU service time. This metric is essentially the load average described in a previous chapter, but modified in order to ensure that its value is always positive and to account for differences in CPU power, if any.

In the remainder of this section we describe the simulation experiments performed. In the first experiment we measured the performance of the load balancing schemes described above, and in the rest of the simulations we explored the effects of various changes in the simulation parameters (such as the amount of local accesses or the number of nodes in the network). The stability of the algorithm in the presence of imperfect load information was also studied.

5.3.1. Performance of the Algorithms

In the first experiment, we simulated a network of five hosts, joined by a CSMA-CD network (see Figure 5.3). For each simulation run, the users in the system were

parameter	value
Nodes in network	5
Network service time	1 Mbit/s (exponentially distributed)
User thinktime	1250 ms (exponentially distributed)
Disk service time	30 ms (exponentially distributed)
CPU service time	10 ms (exponentially distributed)
Message size	1000 bytes
Instructions to process a page	10,000
Fragment size	1-10 pages (uniformly distributed)
Fragments per relation	1
Prob. of local data	0.5, 0.8

Figure 5.2: Parameters for the Model

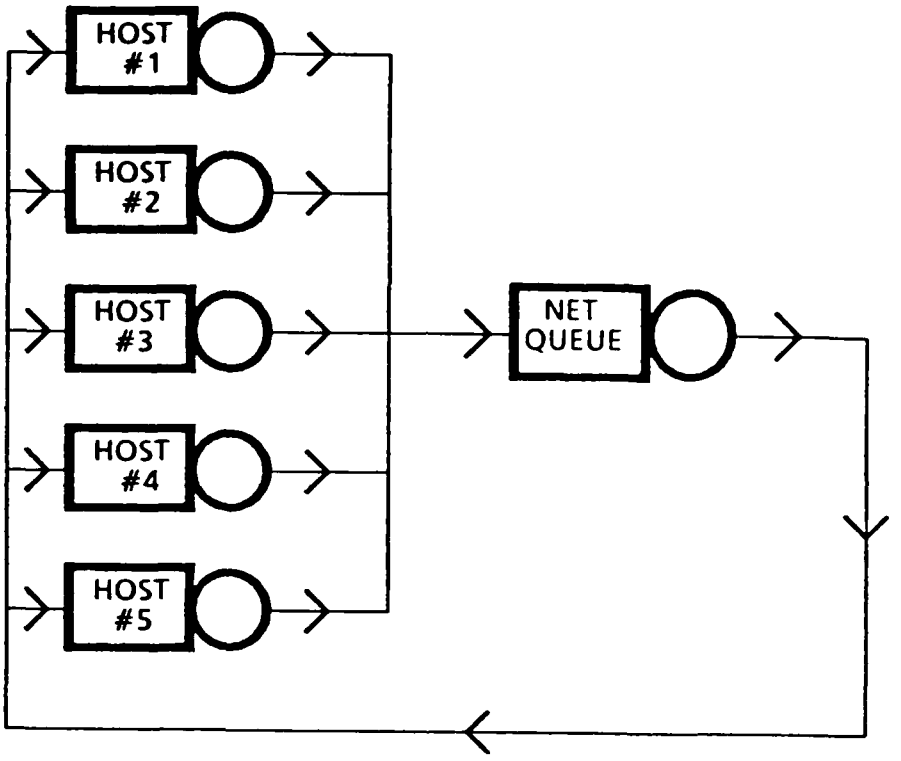


Figure 5.3: Network Model

distributed evenly among three of the processors, while the other two had no users assigned to them (e.g., in the simulation corresponding to 60 system users, three of the processors had 20 users each). However, it should be remembered that, since the data used in a query may be located at any site in the network, work is normally done at all processors, even under the non-load balancing policies FR and RS (work is performed not only for data retrieval purposes, but also in the execution of the local sub-queries).

We varied the number of users (in a single processor) from 15 to 25; thus, the number of total system users varied from 45 to 75. We set the probability of local data

equal to 0.8 (so that 64% of the queries involved only local data, 32% required both local and remote data, and 4% of the queries accessed only remote data). Figure 5.4 shows the average job turn-around time for all the policies, as the number of users in the system increased. The throughput achieved (by the entire system) in the simulations is shown in Figure 5.5.

As the graphs in Figure 5.4 and Figure 5.5 show, LB2 substantially outperformed FR and RS with respect to both performance metrics measured throughout the range of system users we modeled. The added flexibility that LB2 has in selecting the execution sites of the query makes it a very attractive strategy, even under conditions of relatively low load. Furthermore, the network bandwidth was never a bottleneck; its mean utilization never exceeded 10% in our simulation runs. On the other hand, LB1 was not consistently superior to FR and RS until the load in the system became high. We feel that the benefits from LB1 are not sufficient to warrant its use; however, the performance improvement under LB2 was sufficiently high that we would strongly recommend its use in networks where the loads are not completely evenly spread out across the system.

5.3.2. Increased Local Data

In order to explore the effect of a smaller proportion of local data (which, in this situation, implies a decrease in the imbalance of the work demands of the system), we repeated the simulations described above with only 0.5 of the join data being local. The results appear in Figure 5.6 and Figure 5.7. They show that, although the improvement due to LB2 is smaller, it can still be a considerable one (especially in terms of system throughput). We believe that these results support our conviction that, unless the user requests are evenly distributed across the different nodes in the system, load balancing strategies can be very effective in improving DDBMS performance. It should be noted that the environment simulated in our models is one of relatively mild imbalance, at least compared to many of the networks in which we have worked.

It is also interesting to note that FR did not perform demonstrably better than the simpler RS. Thus, we would not recommend that the current design of distributed INGRES be modified in order to implement FR, unless the spread of relation sizes is much greater than the one modeled here.

5.3.3. The Effect of a Larger Network

Since the size of the network modeled was not very large, and we expect newer networks to connect larger numbers of machines, we were interested in determining how an increase in the number of nodes would affect our results. Thus, we repeated the simulations of Section 5.3.1, but this time increased the size of the network from 5 to 10 nodes. The total number of users in the system ranged from 30 to 120, evenly distributed among 6 of the machines, while 4 of the machines had no users.

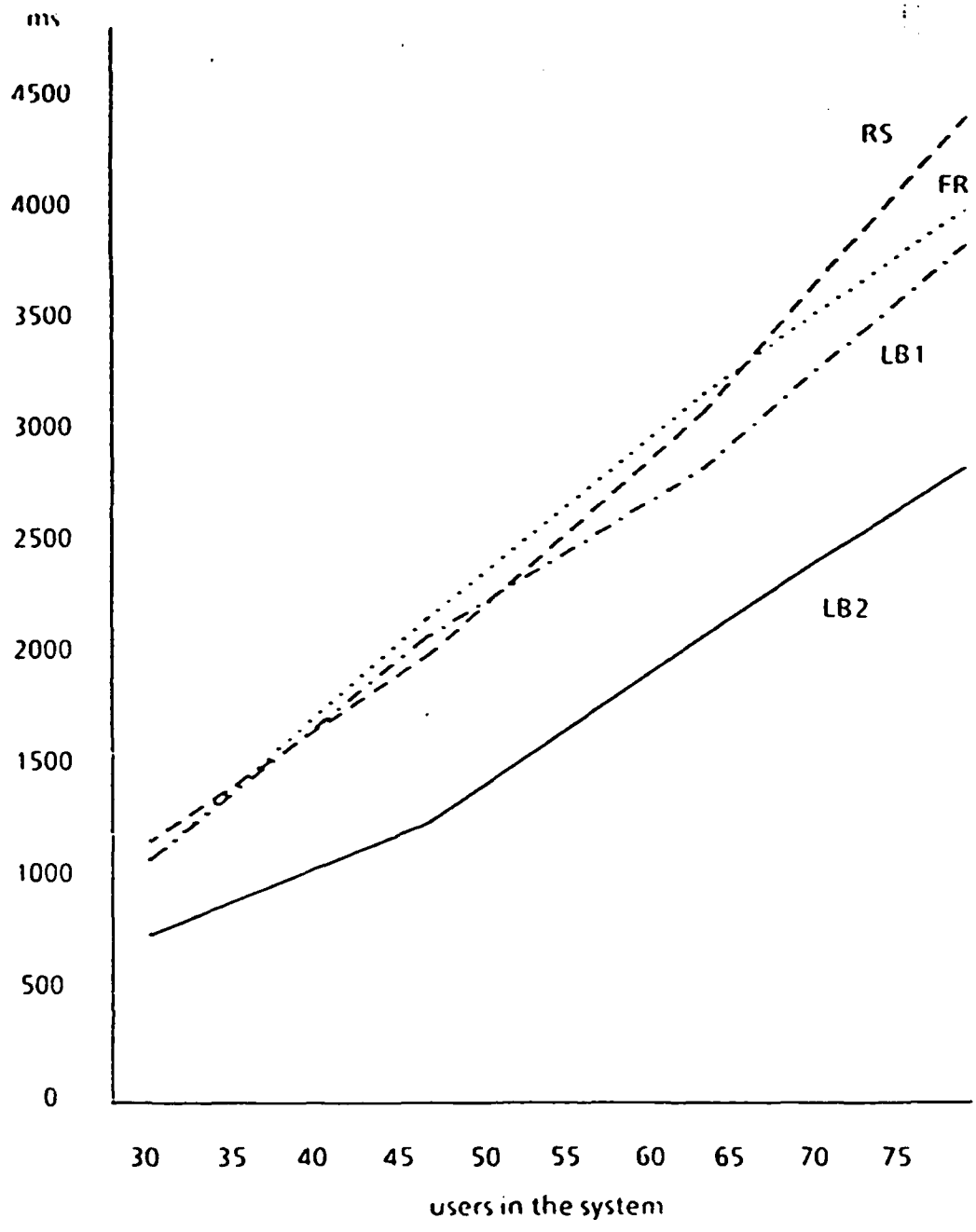


Figure 5.4: Turnaround Time vs. # of Users (0.8 prob. of local data)

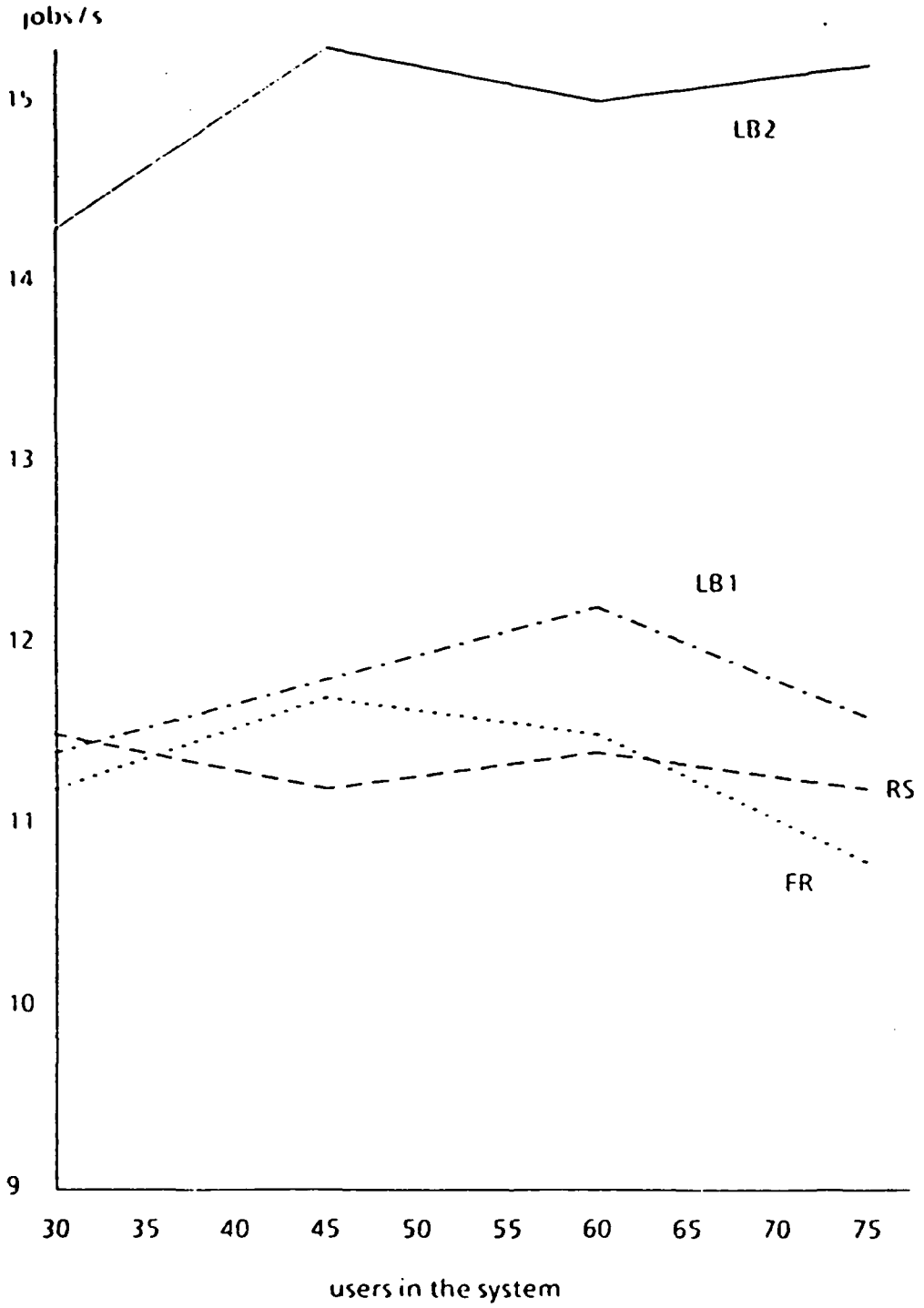


Figure 5.5: Throughput vs. # of Users (0.8 prob. of local data)

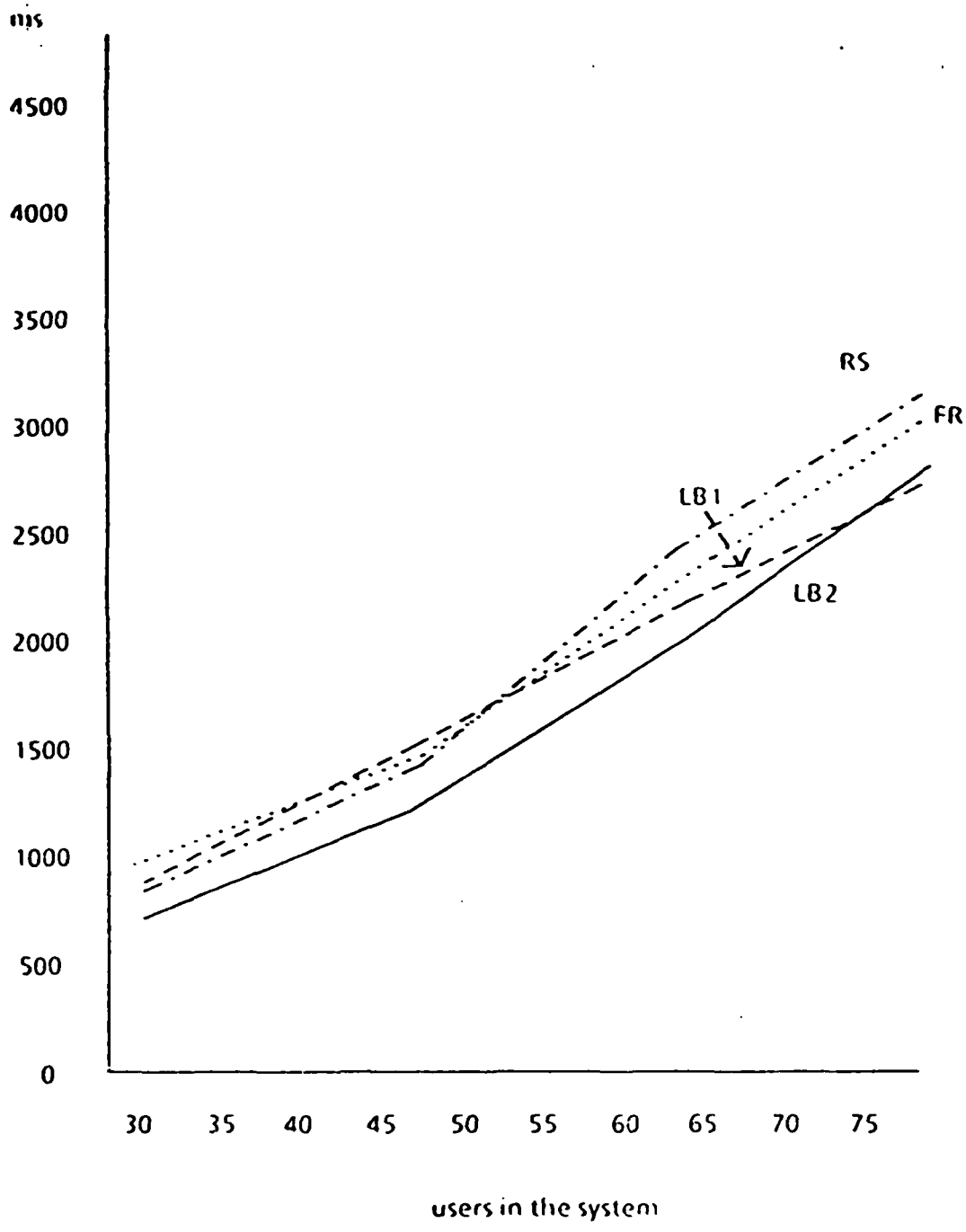


Figure 5.6: Turnaround Time vs. # of Users (0.5 prob. of local data)

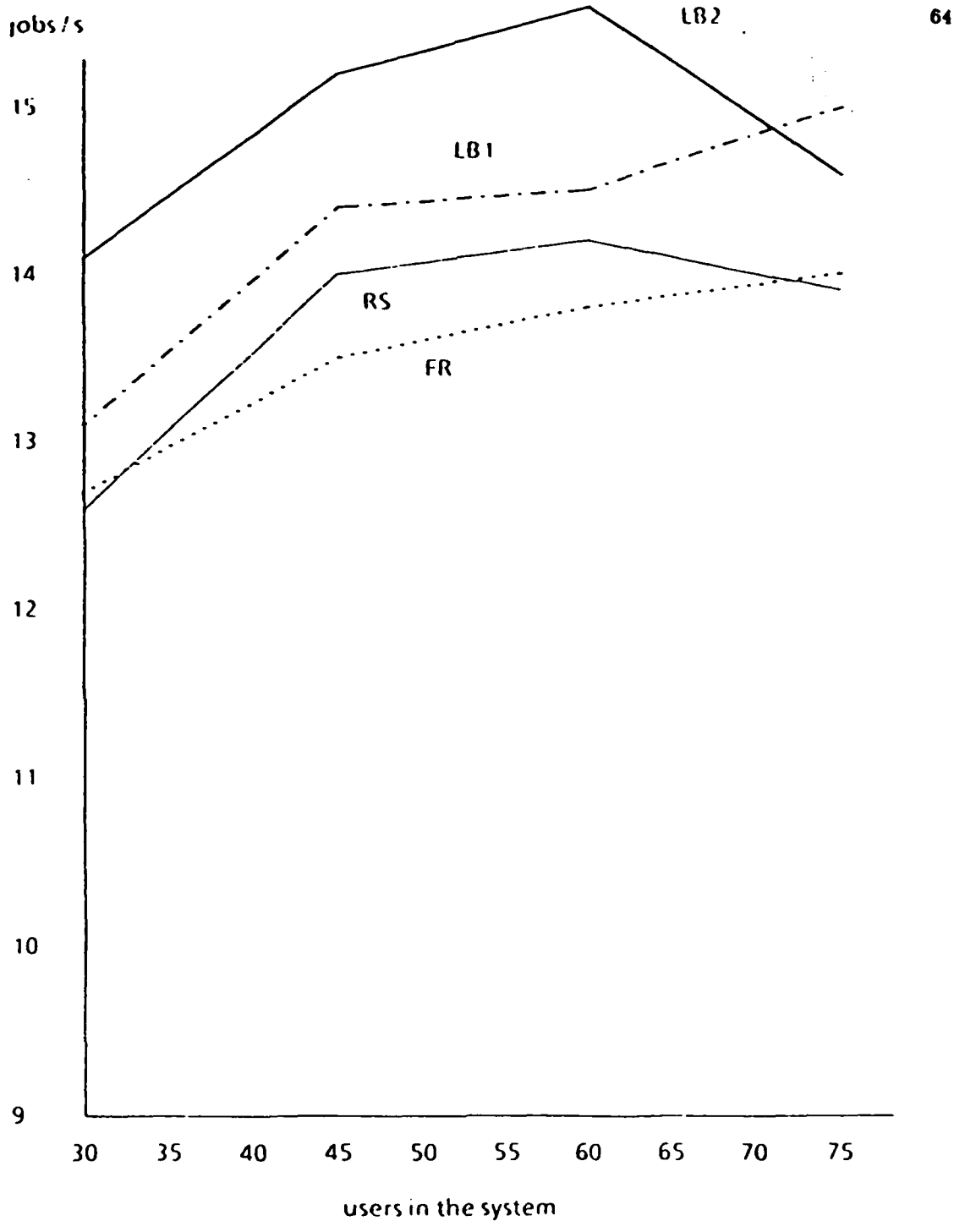


Figure 5.7: Throughput vs. # of Users (0.5 prob. of local data)

While the increased number of processors may provide a greater opportunity for load balancing, since the load balancing decisions are made by the individual processors, one may expect that, with a larger number of decision-makers, more conflicts may arise (i.e., more sites will select the same machine to which to migrate jobs). However, our simulation results show that LB2 continued to offer performance gains over the non-load balancing policies. For example, with 120 users, LB2 offered a response time improvement of more than 35% over the standard fragment-replicate policy (FR). In Figure 5.8 and Figure 5.9 we show, respectively, the mean turnaround time and the throughput corresponding to the various algorithms. It should be pointed out that a network of 10 nodes is not inordinately large; indeed, there are many substantially larger networks in current use, and we expect local area networks to increase in size to much larger numbers over the next few years. On the other hand, even if networks grow to contain hundreds of nodes, perhaps the number of machines that would participate in a load balancing scheme would be a subset of the total. At any rate, the simulation results of this section suggest that our load balancing schemes would still perform well in those larger networks of the near future.

5.3.4. The Effect of Workload Changes

Since the jobs being modeled in our first experiment did not access a large number of database pages (in the previous simulations, each join accessed from 1 to 10 database pages, uniformly distributed) we studied the effect of having more demanding database tasks. In particular, for the simulations described in this section, each join accessed from 1 to 100 database pages, uniformly distributed. All other model parameters are the same as in the first experiment described in this chapter. As can be seen from Figure 5.10 (which shows the throughput of the system for the different algorithms), the increased demands made by the jobs on the system's resources result in their quickly exhausting the capacity of the DDBMS. However, since LB2 can take better advantage of the less loaded processors than the other algorithms, its mean turnaround time (shown in Figure 5.11) is much better than that of the other schemes.

Since we might expect that, even in a well-designed DDBMS, the user demands at some of the nodes might become exceptionally large for short periods of time, a query optimizer that implements LB2 would be very useful, since it would allow the DDBMS to make use of underutilized machines elsewhere in the network, and thus alleviate the temporary crisis.

5.3.5. The Stability of the Algorithm

Simulations were also performed that explored the effect of imperfect or out of date load information on the LB2 policy. The same model (see Figure 5.1) and parameters (see Figure 5.2) employed in the first experiment described above were also used, but now the load information (i.e., the value of the load metric) lagged behind the actual values. The reason for exploring the behavior of LB2 under "stale" data is that, in an

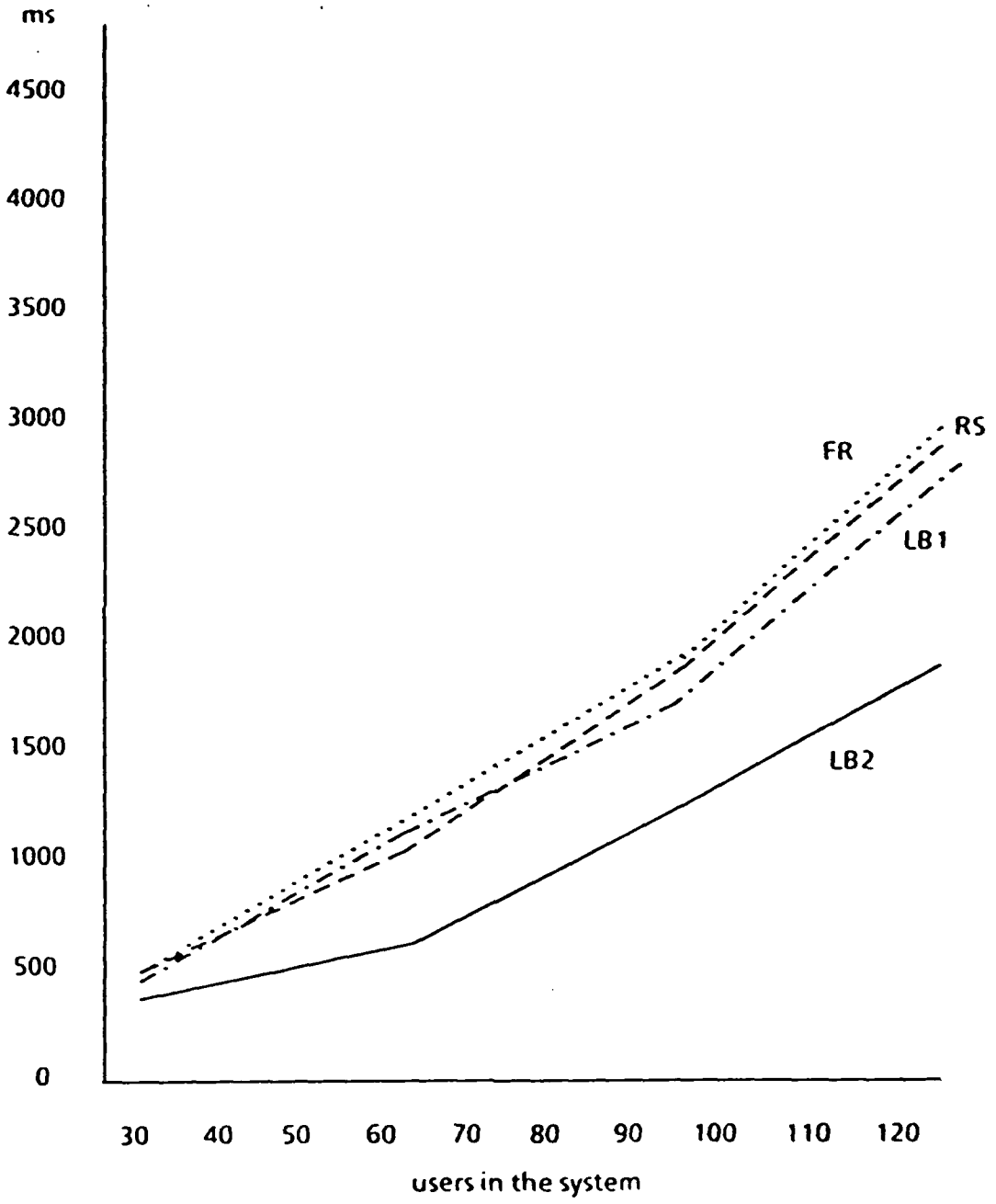


Figure 5.8: Turnaround Time vs. # of Users (10 nodes)

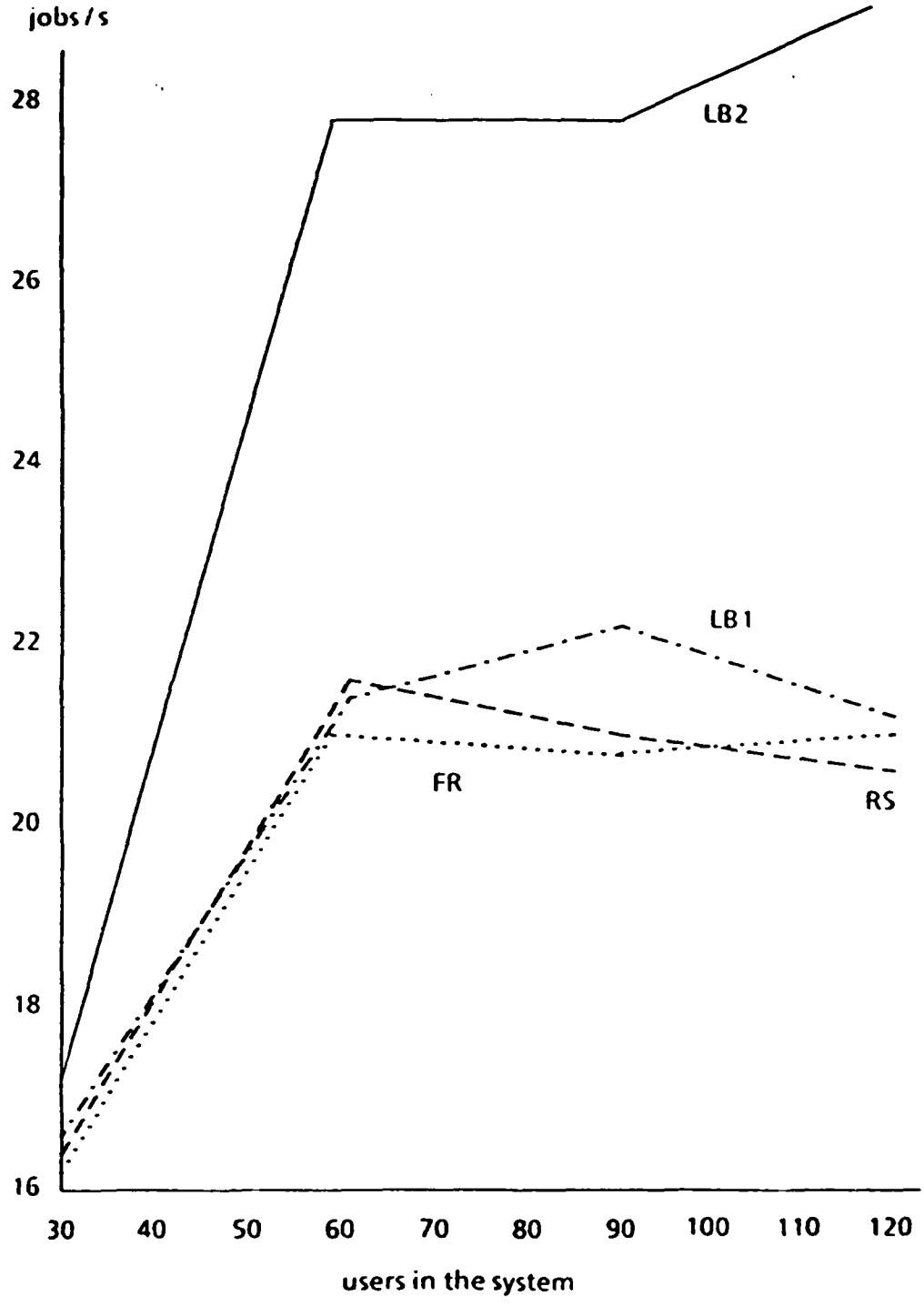


Figure 5.9: Throughput vs. # of Users (10 nodes)

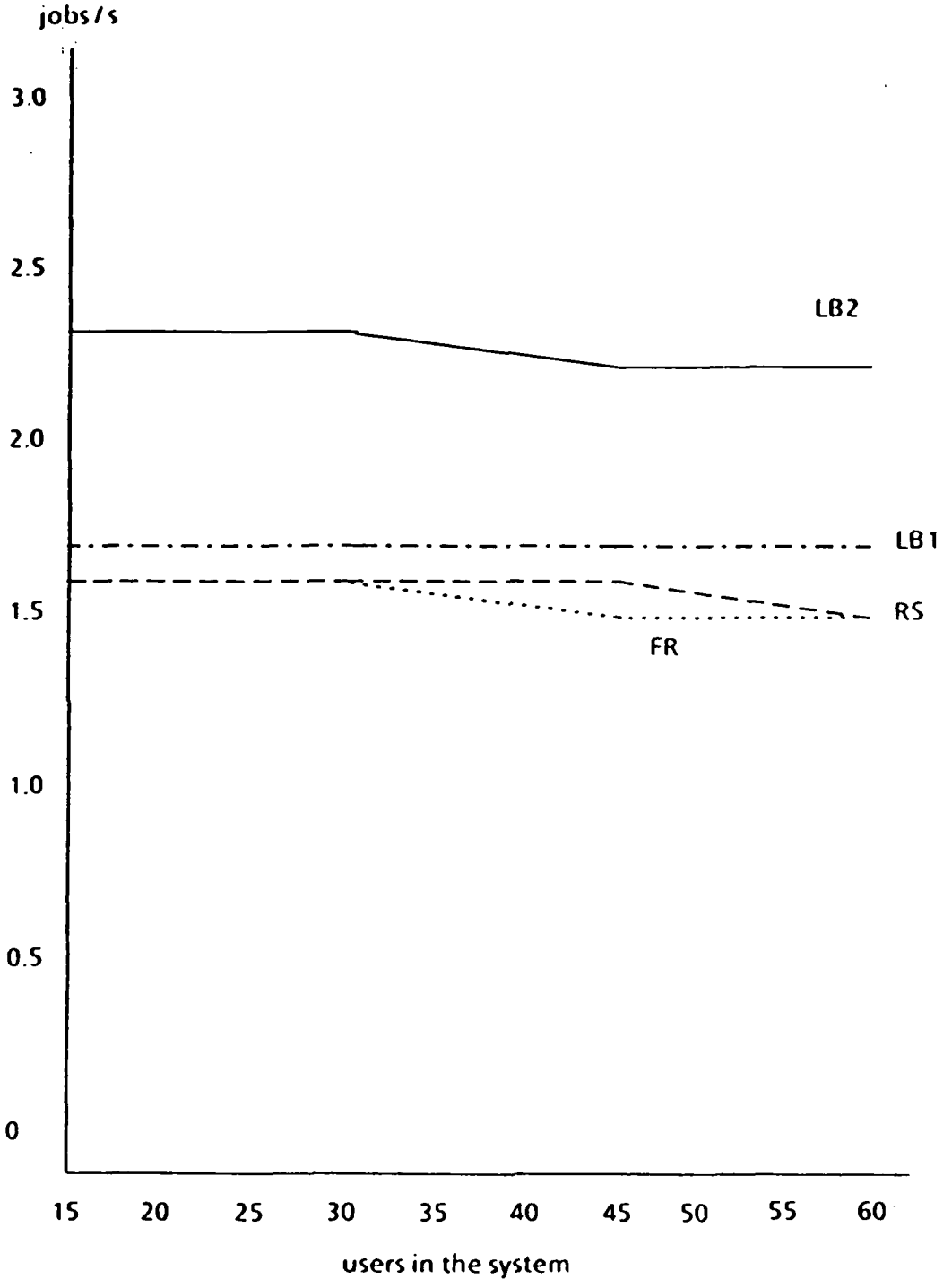


Figure 5.10: Throughput vs. # of Users (increased workload)

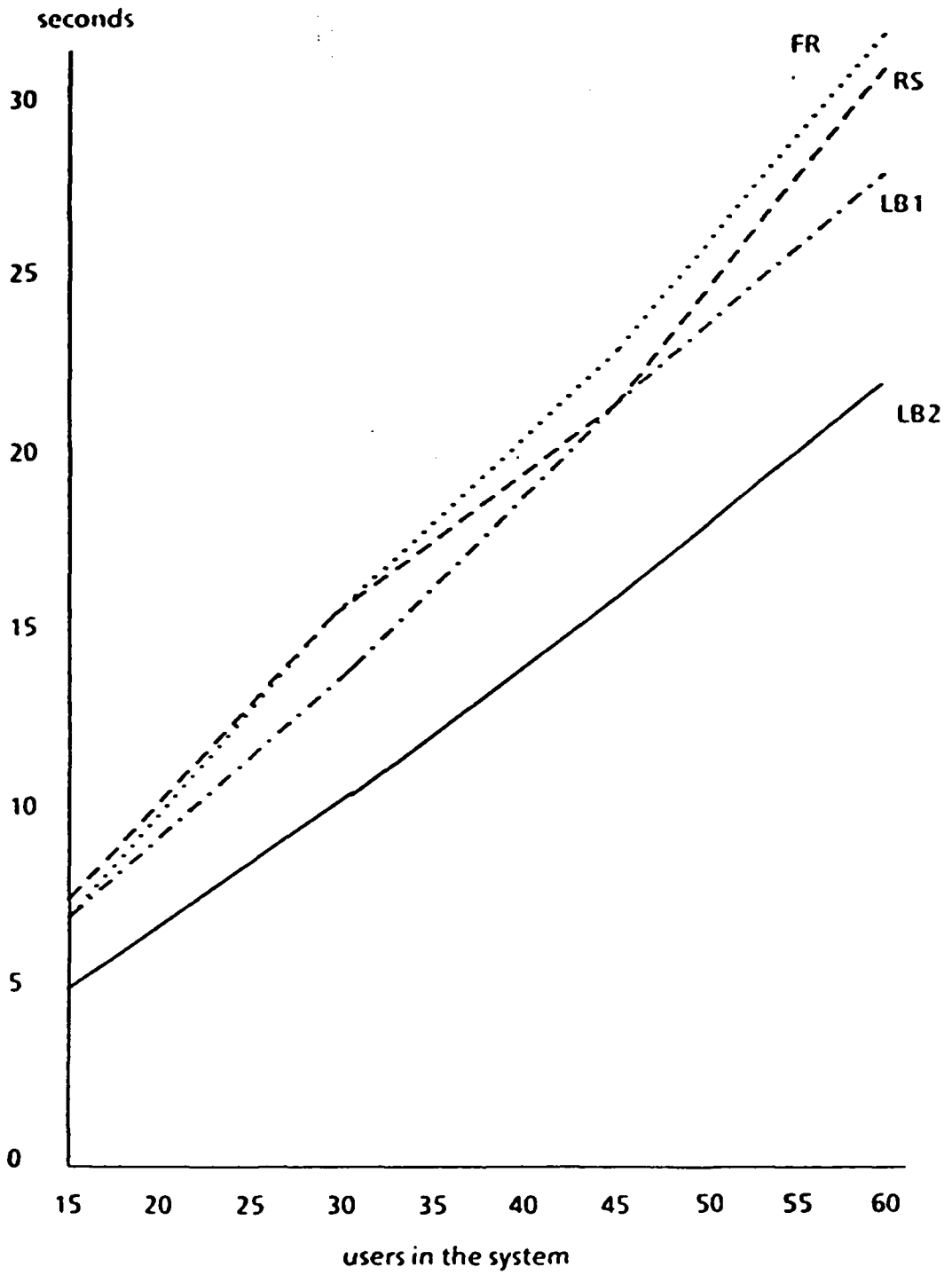


Figure 5.11: Turnaround Time vs. # of Users (increased workload)

actual implementation, there is a certain cost to sampling and broadcasting the load information. Thus, we are interested in the performance of LB2 as we let the load information used for the load balancing decision become increasingly out of date. As in the simulations presented in Chapter 3, this situation was modeled by periodically refreshing each processor's estimates of the load present at other nodes. Clearly, for a given load metric update rate, as the system load increases, we experience a degradation in performance with respect to the results obtained for LB2 with up to date information.

For this experiment, the number of users in the system was set at 45. The update interval was varied from 0 to 7000 milliseconds. Figure 5.12 shows the effect of decreasing the refresh rate on the mean turn-around time experienced by jobs in the system. In order to understand the significance of the graph better, an additional line is shown in the figure; a hatched line marks the mean turn-around time for jobs in the system if no load balancing takes place (i.e., if the FR policy is used). As is evident from the graph, as the update interval increases, the performance of the system begins to degrade from its best value (with perfect information), until, finally, the mean turn-around time gets worse with load balancing than without it. Figure 5.13 shows that a similar result holds for the system throughput.

The results of this last experiment show that, in order to successfully implement LB2, load information must be broadcast at a fairly rapid rate. For a system similar to the one being modeled, the refresh interval must be smaller than 2 seconds in order for LB2 to be an effective strategy. Clearly, the length of this interval depends (in a non-trivial way) on the number of users currently in the system, and on the types of jobs they are executing.

It is interesting to investigate what happens as the number of users in the system increases. While more jobs will enter the system during the update interval (and thus there will be greater opportunity for erroneous selection of the fragmented relation), we also expect load balancing to be more effective when the load is higher. We repeated the above experiment, but this time with 60 users in the system. Figure 5.14 and Figure 5.15 show (respectively) the effect of this change on the turnaround time and on the throughput. The constraints on the update rate are somewhat relaxed; we can now wait 3 seconds between updates and still improve the performance of the system (as compared to FR).

As we mentioned above, there is a cost involved in rapidly communicating load information among the network nodes. This cost can be divided into two parts: (1) the expense of obtaining the load information at each node, and (2) the cost of broadcasting that information. Of these two costs, the second is the more sizable one, and the one that grows in proportion to the number of hosts on the network (even if the network supports a broadcast facility, so that one does not have to send a message to every other node, there is a cost associated to listening to the messages from every other site). This suggests that perhaps an acceptable compromise between paying the high cost of using recent data and suffering the performance penalty due to stale data might be to obtain

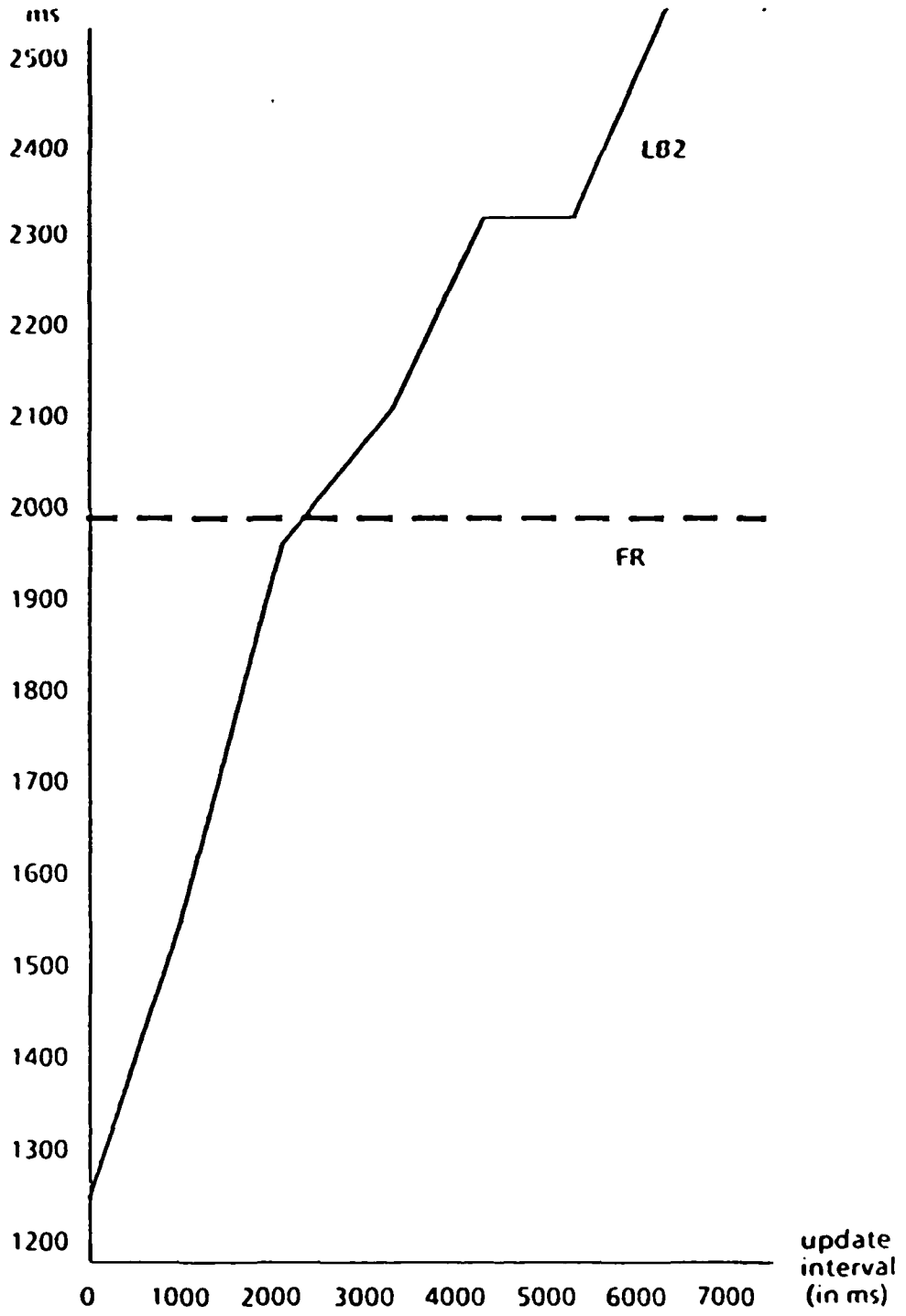


Figure 5.12: Turnaround Time vs. Update Rate (45 users)

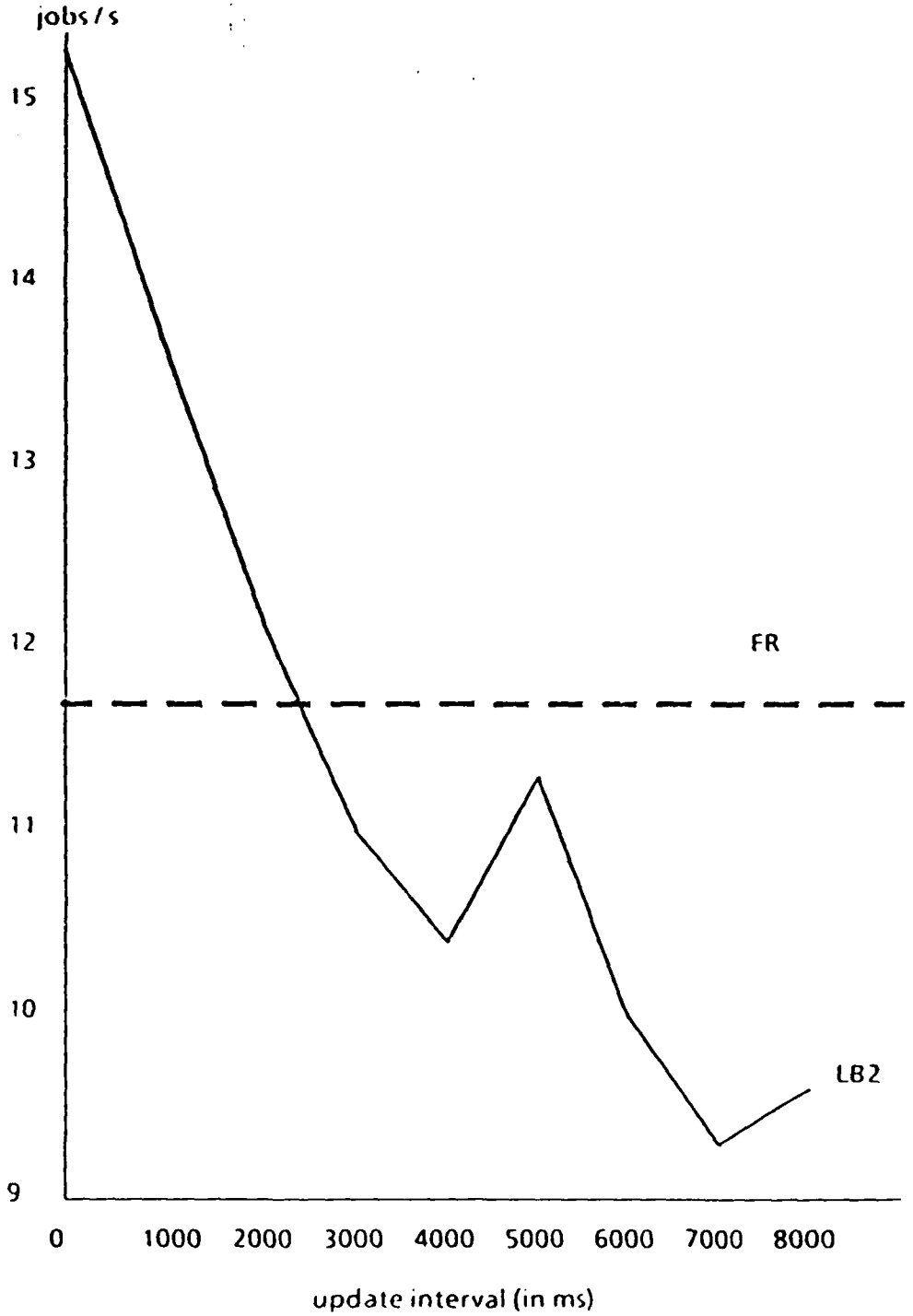


Figure 5.13: Throughput vs. Update Rate (45 users)

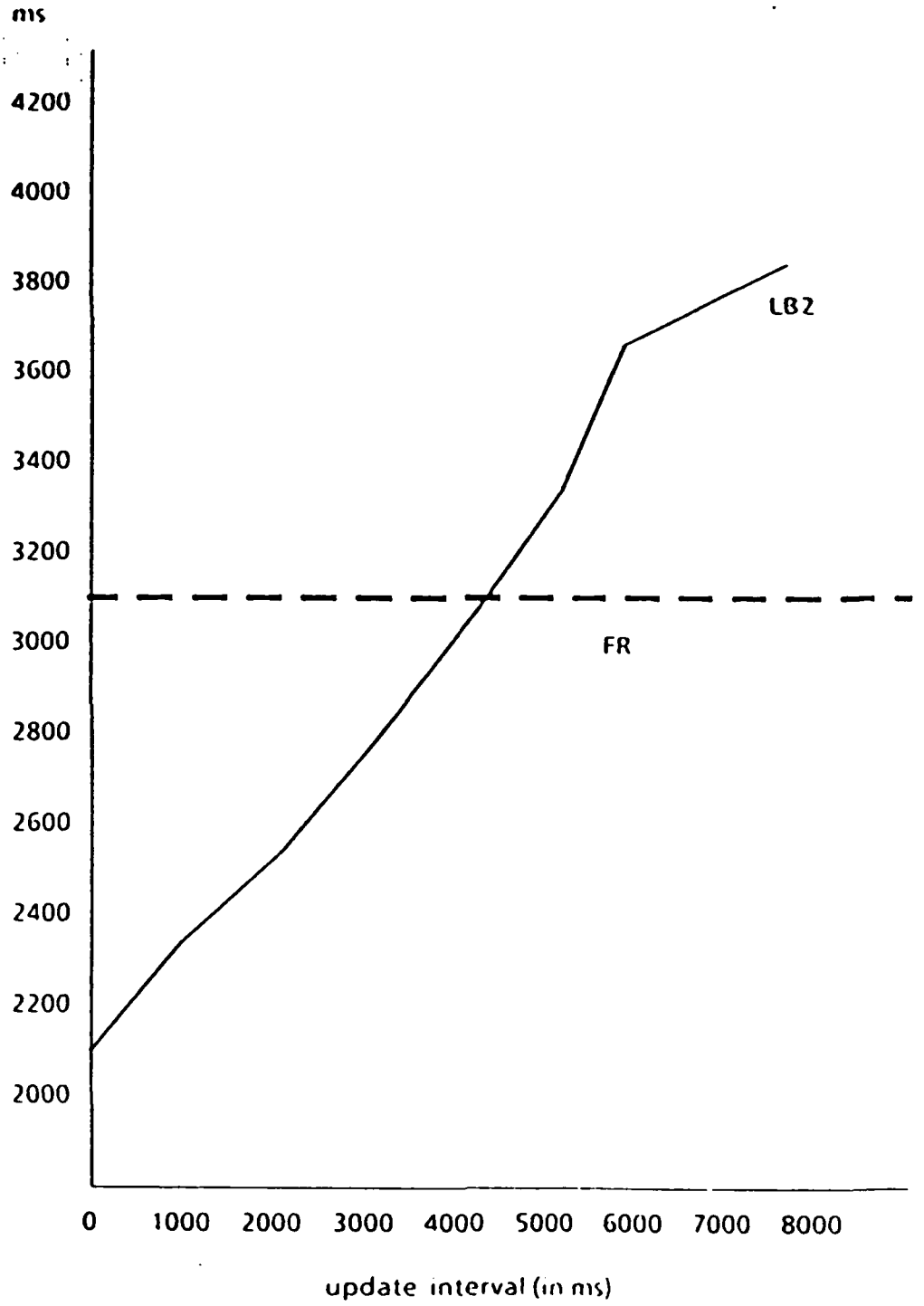


Figure 5.14: Turnaround Time vs. Update Rate (60 users)

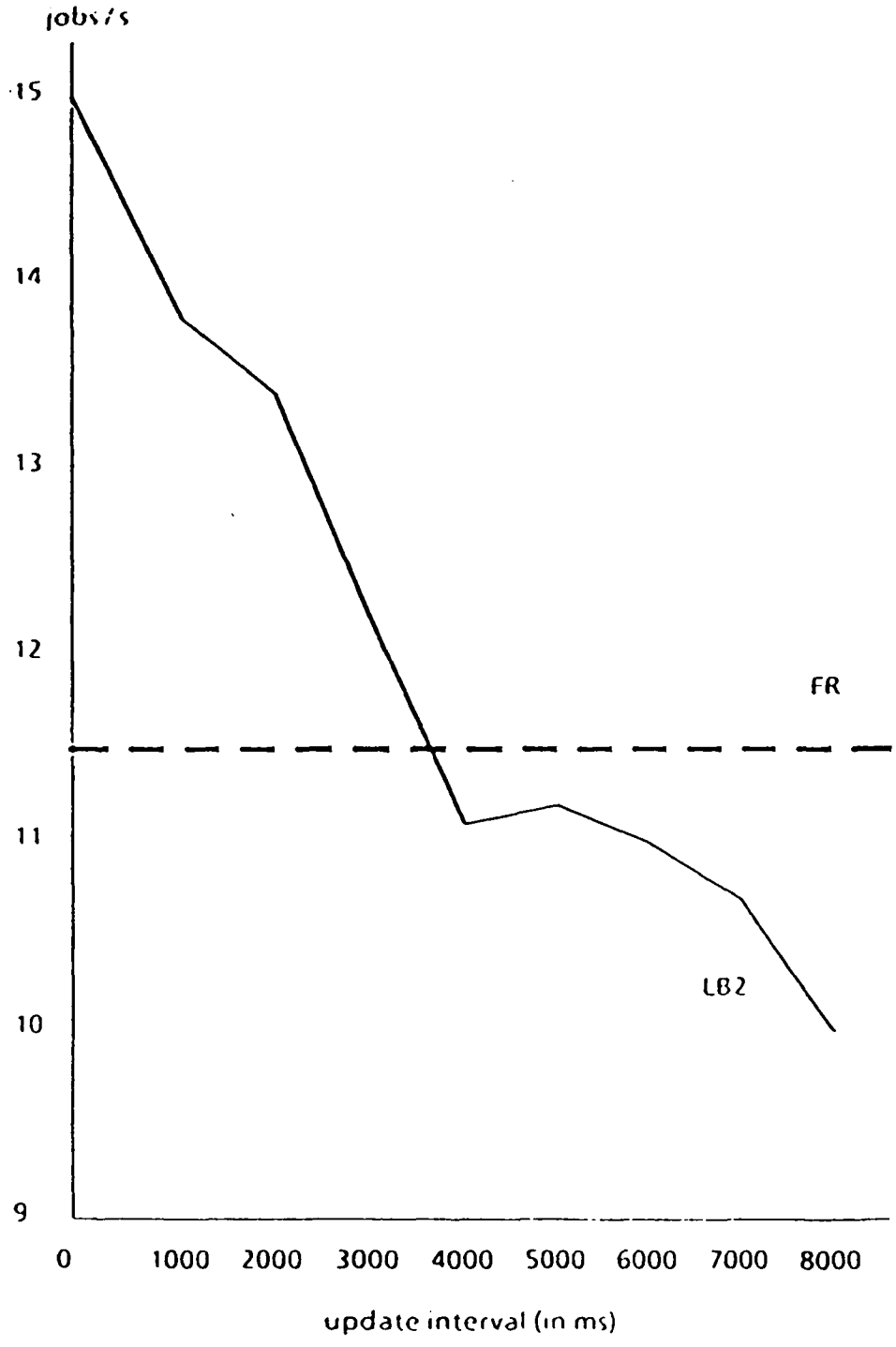


Figure 5.15: Throughput vs. Update Rate (60 users)

the local load information at very frequent intervals, but to broadcast that information at a much slower rate. The query optimizer would still have to cope with stale information about the state of the other processors in the network, but would at least be able to assess its own load more accurately.

In the final experiment described in this chapter, we studied the situation described in the previous paragraph; we used the same model and parameters as in the first model of this sub-section, but this time we used current local load information and stale remote load data. In Figure 5.16 we show the average turnaround time for this simulation. As before, the hatched line marks the performance of the FR policy; the line labeled "LB2-s" shows the performance of LB2 with completely stale data (i.e., it is the same line that appears in Figure 5.12); the line marked "LB2-l" corresponds to running LB2 with current local load data. Similarly, by examining Figure 5.17 we can compare the throughput curves of LB2-s and LB2-l.

As both Figure 5.16 and Figure 5.17 show, the behavior of LB2 (i.e., LB2-l) becomes much more stable now. Although the performance of the algorithm is somewhat degraded from that of the perfect information case, LB2 performs better than the non-load balancing case even for fairly slow rates of load broadcast. The instability that LB2 showed in the previous two experiments was due to two causes: (1) each node had old information about remote loads, and thus, reacted slowly to changes in the load of other sites, and (2) the machines tended to keep for themselves all local work, even past the point when they were overloaded, since the local load information also lagged behind. By eliminating the second factor, the performance of LB2 improved radically. This suggests that, for an implementation of LB2, one would not have to incur the costs of constantly communicating load information in order to achieve performance gains.

Even though the availability of current local load information helped matters a great deal, the comments we made in Chapter 3 about the significance of the update rate for load balancing still apply. In order for load balancing techniques to be effective, load metric information must be rapidly communicated. In practice, this may turn out to impose a limit on the number of processors that can usefully participate in a load balancing arrangement. When the distributed system has too many machines, a possible solution is to logically decompose a network into clusters of processors and only implement intra-cluster load balancing.

It should be kept in mind that the results of the simulations presented in this chapter underestimate the performance improvements possible while using load balancing, since the models used do not account for the non-linear effects of load on CPU performance (i.e., five jobs do not take five times as much to complete as a single one, but much more, due to scheduling overhead and other such effects not modeled in our simulation). Thus, the benefits of LB1 and LB2 may be even greater in an actual implementation than what our results suggest.

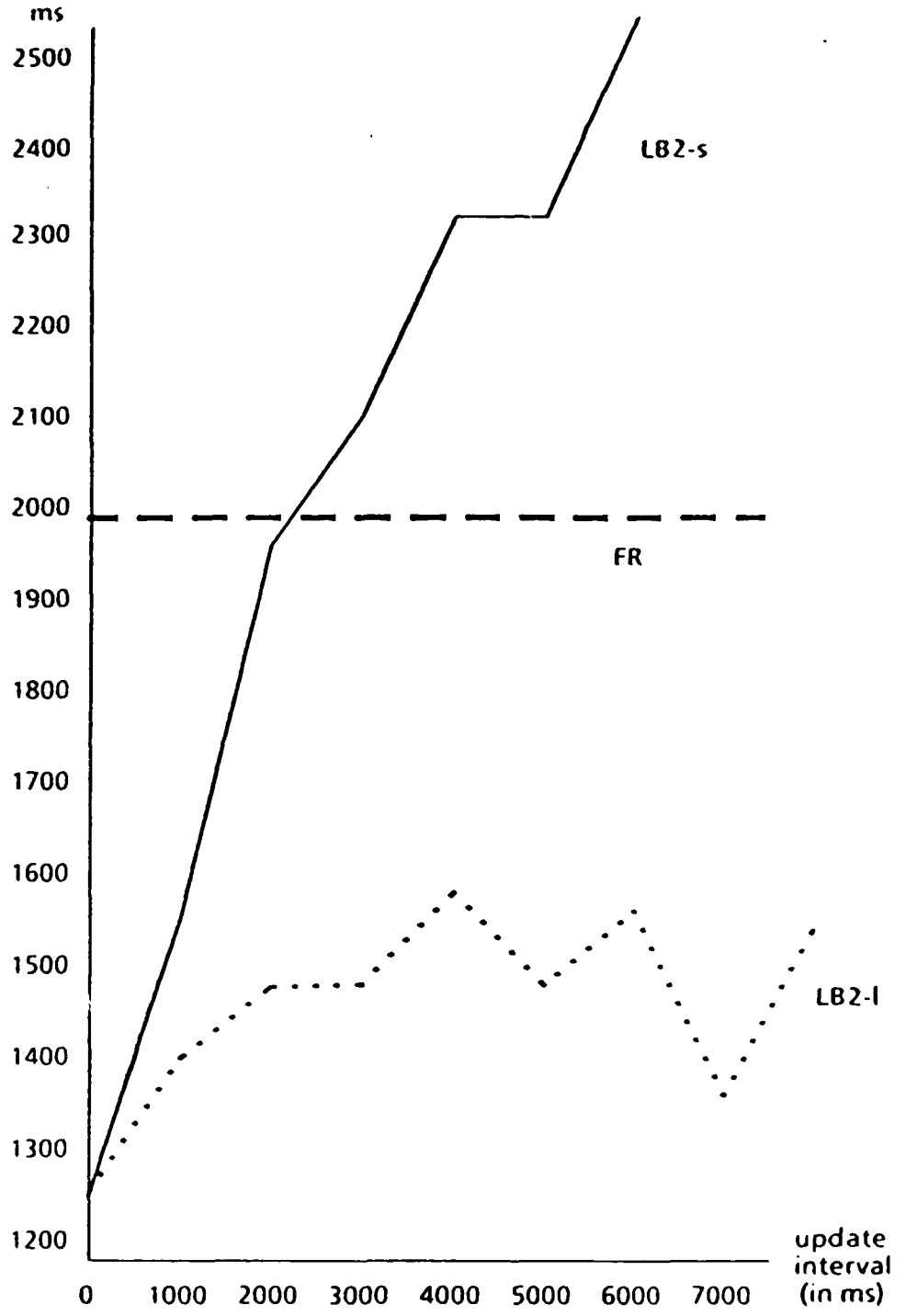


Figure 5.16: Turnaround Time vs. Update Rate (current local data)

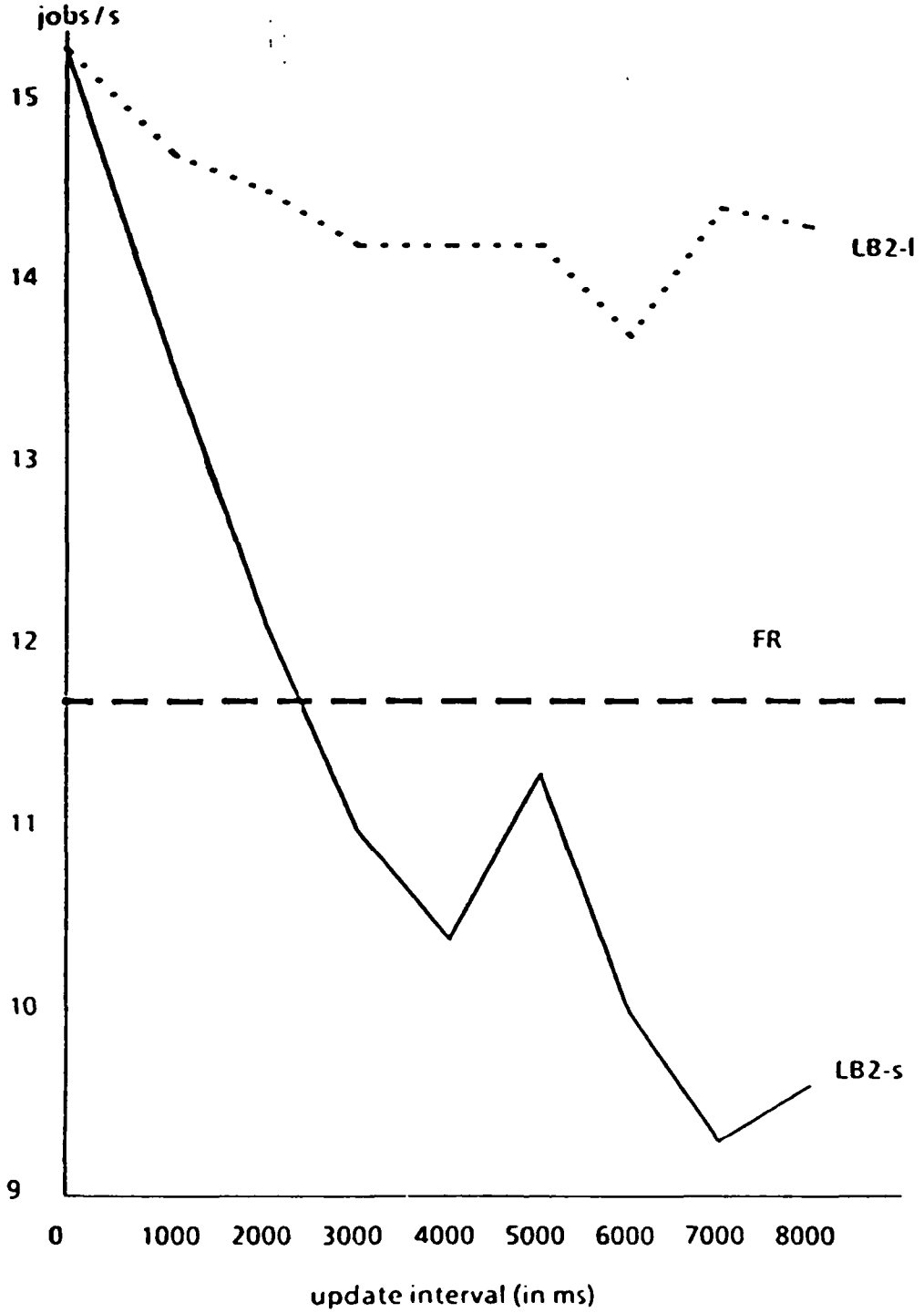


Figure 5.17: Throughput vs. Update Rate (current local data)

5.4. Conclusions

In this chapter, we saw that, for distributed INGRES, a second DDBMS quite different from R* (which we studied in the previous chapter), we can also obtain sizable performance improvements by applying load balancing techniques to the query optimization software. This lends further credibility to our claim that load balancing techniques can be profitably introduced into DDBMS optimizers.

In the next and final chapter, we will make some overall comments about the significance and usefulness of our work, and map future directions for research in this area.

CHAPTER 6

Conclusions and Future Work

6.1. Conclusions

In order to gauge the success of the research we have performed, the goals stated at the outset of the work must be recalled. In the introductory chapter of this thesis, the following issues were raised:

> can the performance of a DDBMS be improved by using load balancing techniques? It is clear that some degree of improvement could always be achieved by giving more knowledge about the system to the query planner, but only if the gains are sizable will this proposal be of more than academic interest. Note that system designers, who have sunk many person-years of effort into their query optimizers, might be reluctant to discard their current software and develop new code that incorporates load balancing. Thus, it would be desirable to reduce to a minimum the changes the introduction of a load balancing scheme will require. 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. Hence, a better statement of the problem would be: can load balancing strategies be easily introduced into current optimizers in such a way that the performance of the system will be substantially improved in realistic situations of imbalance and under reasonable conditions?

In the previous chapters, two different systems were studied, one experimentally and the other by simulation. For both R* and distributed INGRES it was shown that the gains due to load balancing could indeed be large even under moderate load imbalances, and for systems that were never saturated. For example, in the R* experiment it was shown that for a typical join the gains were from about 10 to 30 percent, in a system that never became very loaded (always less than 90 percent CPU utilization). For both systems the implementation approaches suggested involved minimal changes to the system. In the case of INGRES, the cost function was extended so that, instead of examining only transmission costs in order to make the fragment-replicate decision, the load average measurement gathered by UNIX systems was also considered. For R*, one of the two implementations described was to construct alternative plans and choose among them based on load information. The other approach was to invalidate plans at run-time if the load in the systems involved was not within a "normal" range. Finally, in all cases the load metrics used were readily available in the systems studied (although we do not claim that those metrics are necessarily good ones; see [Ferrari1985] for some recent work on this subject). Since the statistics used are normally gathered by the operating systems, implementing load balancing does not involve additional overhead for

data acquisition, but only for communication among the systems involved. In a high bandwidth broadcast local area network such as the Ethernet, this communication overhead should be negligible unless an extraordinarily large number of hosts are involved in the load balancing scheme.

The results described in this thesis show that load balancing strategies are quite attractive for distributed database systems. Furthermore, designers would do well to focus on runtime issues such as the loads, even at the expense of doing less work on compile time optimization. The effect of the former has been shown here to be significant, while any new gains due to further refinements in the techniques currently employed for query compilation might be quite marginal.

6.2. Future Work

The work presented in this thesis could be extended in a variety of ways. One possibility would be to focus next on non-relational distributed databases (for example, a hierarchical database system such as IMS [Date1975]). Another interesting project would be to actually implement some of the load balancing algorithms studied, either in INGRES or in R*, and to measure the systems under actual workloads. A third possibility is to use the experience obtained from the work herein to attack the general load balancing problem, i.e., to study the problem in the context of a general purpose distributed system. As has been mentioned in previous chapters, the load balancing work described in the published literature suffers from a number of deficiencies (e.g., the policies are too *ad hoc*, or the models used are unrealistic). One possible approach would be to study a collection of load metrics in order to determine their suitability. Since the choice of metric would seem to be very system and workload dependent, a choice of both would have to be made. Berkeley UNIX 4.2 BSD and the workload of one of the distributed systems at U.C. Berkeley would seem to be appropriate initial choices. Once the choice of a metric has been narrowed down to a few, a variety of load balancing policies could be explored. The policies should be chosen based on both intuition and the predictions of theoretical models. The study and selection of both load metrics and load balancing policies could be done by constructing simulation models of the existing software and hardware at Berkeley and by performing experiments on the campus systems. One of the results of this line of research should be an implementation of load balancing suitable, at least, for the environments where 4.2 BSD is typically run (i.e., university, research laboratory, and engineering design installations). With careful extensions, this work could also result in even more general strategies, which would be applicable in a wider setting.

BIBLIOGRAPHY

[Almes1979]
Almes, Guy T. and Lazowska, Edward D., "The Behavior of Ethernet-Like Computer Communications Networks," *Proceedings 7th ACM Symposium on Operating Systems Principles, Operating Systems Review*, pp. 66-81 (December 1979).

[Alonso1982]
Alonso, Rafael, "A Model of the XCS Network," CS258 class report, U.C. Berkeley (Winter 1982).

[Astrahan1976]
Astrahan, M., Blasgen, M. W., Chamberlin, D. D., Eswaran, K. P., Gray, J. N., Griffiths, P. P., King, W. P., Lorie, R. A., McJones, P. R., Mehl, J. W., Putzolu, G. R., Traiger, I. L., Wade, B. W., and Watson, V., "System R: Relational approach to database management," *ACM Transactions on Database Systems* **1**(2) pp. 97-137 (June 1976).

[Birrell1982]
Birrell, A., Levin, R., Needham, R. M., and Schroeder, M. D., "Grapevine: An Exercise in Distributed Computing," *Communications of the ACM* **25**(4) pp. 260-274 (April 1982).

[Blasgen1979]
Blasgen, M. et al, "System R: An architectural update," Research Report RJ2581, IBM Research Laboratory, San Jose, California (July 1979).

[Bokhari1979]
Bokhari, S. H., "Dual Processor Scheduling with Dynamic Reassignment," *IEEE Transactions on Software Engineering* **SF-5**(4) pp. 341-349 (July 1979).

[Bokhari1981]
Bokhari, S. H., "A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in a Distributed Processor System," *IEEE Transactions on Software Engineering* **SE-7**(6)(November 1981).

[Bryant]
Bryant, R. M., *SIMPAS 5.0 User Manual*, University of Wisconsin - Madison ().

[Bryant1980]
Bryant, R. M., "SIMPAS - A Simulation Language Based on Pascal," *Proceedings of the 1980 Winter Simulation Conference*, pp. 25-40 (December 1980).

[Bryant1981]
Bryant, R. M. and Finkel, R. A., "A Stable Distributed Scheduling Algorithm," *Proceedings of Second International Conference on Distributed Computing Systems, Paris*, pp. 314-323 (1981).

[Ceri1982]
Ceri, S. and Pelagatti, G., "Allocation of Operations in Distributed Database Access," *IEEE Transactions on Computers* **c-31**(2)(1982).

[Chamberlin1974]
Chamberlin, D. D. and Boyce, R. F., "SEQUEL: A Structured English Query Language," *Proc. of the ACM-SIGMOD Workshop on Data Description, Access and Control*, (May 1974).

[Cheriton1983]
Cheriton, D. R. and Zwaenepoel, W., "The Distributed V Kernel and its Performance for Diskless Workstations," *ACM Operating Systems Review* **17**(5) pp. 128-139 (October 1983). Proceedings Ninth ACM Symposium on Operating Systems Principles

[Chou1982]
Chou, T. C. K. and Abraham, J. A., "Load Balancing in Distributed Systems," *IEEE Transactions on Software Engineering* **SE-8**(4)(July 1982).

[Chow1977]
Chow, Y. C. and Kohler, W. H., "Dynamic Load Balancing in Homogeneous Two-Processor Distributed Systems," *Proceedings of International Symposium on Computer Performance Modeling, Measurement and Evaluation, Yorktown Heights, NY*, pp. 39-52 (August 1977). also in Computer Performance, North Holland

[Chow1979]
Chow, Y. C. and Kohler, W, H., "Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System," *IEEE Transactions on Computers* **C-28**(5) pp. 354-361 (May 1979).

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

[Chuanshan1984]
Chuanshan, Gao, Liu, Jane W. S., and Railey, Malcolm, "Load Balancing Algorithms in Homogeneous Distributed Systems," DRAFT - Department of Computer Science, University of Illinois (1984).

[Clark1978]
Clark, D. D., Pogran, K. T., and Reed, D. P., "An Introduction to Local Area Networks," *Proceedings of the IEEE* **66**(11) pp. 1497-1517 (November 1978).

[Coffman1977]
Coffman, E. G., Leung, J. Y-T., and Ting, D., "Bin-Packing Problems and their Applications in Storage and Processor Allocation," *Proceedings of International Symposium on Computer Performance Modeling, Measurement and Evaluation, Yorktown Heights, NY*, pp. 327-339 (August 1977). also in Computer Performance, North Holland

[Cook1983]
Cook, R., Finkel, R., DeWitt, D., Landweber, L., and Virgillio, T., "The Crystal Nugget - Part I of the First Report on the Crystal Project," Computer Sciences Technical Report #499, University of Wisconsin - Madison (April 1983).

[Cook1983]
Cook, R., Finkel, R., Gerber, B., DeWitt, D., and Landweber, L., "The Crystal Nuggetmaster - Part II of the First Report on the Crystal Project," Computer Sciences Technical Report #500, University of Wisconsin - Madison (April 1983).

[Daniels1982]
Daniels, Dean, "Query Compilation in a Distributed Database System," Research Report RJ3423 (40689) 3/22/82, IBM San Jose Research Lab (1982).

[Date1975]
Date, C. J., "An Introduction To Database Systems," *Addison-Wesley*, (1975).

[Denning1968]
Denning, P. J., "Thrashing: Its Causes and Prevention," *Proc. AFIPS FJCC, Pt 1* **33** pp. 915-922 AFIPS Press, Montvale NJ, (1968).

[Epstein1978]
Epstein, R., Stonebraker, M. R., and Wong, E., "Distributed Query Processing in a Relational Data Base System," Memorandum No. UCB/ERL M78/18, Electronics Research Laboratory, U.C. Berkeley (17 April 1978).

[Ferrari1978]
Ferrari, Domenico, "," in *Computer Systems Performance Evaluation*, Prentice-

Hall, Inc. (1978).

[Ferrari1983]
Ferrari, Domenico, "The Evolution of Berkeley Unix," Report No. UCB/CSD 83/155, U.C. Berkeley (December 1983). also in Proceedings of COMPCON, Spring 1984

[Ferrari1985]
Ferrari, Domenico, "A Study of Load Indices for Load Balancing Schemes," Report No. UCB/CSD 86/262, U.C. Berkeley (October 1985).

[Finkel1984]
Finkel, Raphael, *private communication*, University of Wisconsin-Madison (1984).

[Hagmann1983]
Hagmann, Robert B., "Performance Analysis of Several Backend Architectures," U.C.B. Report No. 83/124 (August 1983).

[Hunter1984]
Hunter, Ed, *private communication*, U.C. Berkeley - Computer Systems Research Group (1984).

[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).

[IBM]
IBM,, "CICS/VS General Information Manual," IBM Form No. GH24-5013 ().

[Jain1978]
Jain, Rajendra K., "Control Theoretic Approach to Computer Systems Performance Improvement," *Proceedings of Computer Performance Evaluation Users Group, Boston*, (1978).

[Kratzer1980]
Kratzer, A. and Hammerstrom, D., "A Study of Load Levelling," *Proceedings of Compcon80: Distributed Computing*, (Fall 1980).

[Krueger1984]
Krueger, Phillip and Finkel, Raphael, "An Adaptive Load Balancing Algorithm for a Multicomputer," Computer Sciences technical Report, University of Wisconsin - Madison (April 1984).

[Lampson1983]
Lampson, Butler W., "Hints for Computer System Design," *Proceedings 9th ACM Symposium on Operating Systems Principles, Operating Systems Review* **17**(5)(October 1983).

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

[Leffler1983a]
Leffler, S., Joy, W., and McKusick, K., *UNIX Programmers's Manual - 4.2 Berkeley Software Distribution,* U.C Berkeley (August 1983).

[Leffler1983b]
Leffler, S. J., Fabry, R. S., and Joy, W. N., "A 4.2 BSD Interprocess Communication Primer," Report No. UCB/CSD 83/145, Computer Science Division, University of California, Berkeley (Draft of July 27, 1983).

[Leffler1983c]
Leffler, S. J., Joy, W. N., and Fabry, R. S., "4.2 BSD Networking Implementation Notes," Report No. UCB/CSD 83/146, Computer Science Division, University of California, Berkeley (Revised July 1983).

[Levy1982]
Levy, S. Y., "Distributed Computation for Design Aids," *Proceedings of the Nineteenth Design Automation Conference, Las Vegas,* (June 14-16 1982).

[Livny1983]
Livny, Miron, "The Study of Load Balancing Algorithms for Decentralized Distributed Processing Systems," Ph.D. Thesis, Weizmann Institute of Science (August 1983).

[Livny1982]
Livny, M. and Melman, M., "Load Balancing in Homogeneous Brodcast Distributed Systems," *Proceedings Computer Network Performance Symposium*, pp. 47-55 (April 1982).

[Lo1981]
Lo, Virginia and Liu, Jane W. S., "Task Assignment in Distributed Multiprocessor Systems," *Proceedings Tenth International Conference in Parallel Processing*, pp. 358-360 (August 1981).

[Ma1982]
Ma, P. R., Lee, E. Y. S., and Tsuchiya, M., "A Task Allocation Model for Distributed Computing Systems," *IEEE Transactions on Computers* **C-31**(1)(January 1982).

[McGrath1983]
McGrath, Sean and Peters, Dick, "Single System Image Software," Computing Services Newsletter, University of California, Berkeley (September 1983).

[McQuillan1977]
McQuillan, J. M. and Walden, D. C., "The ARPA Network Design Decisions," *Computer Networks* **1**(5) pp. 243-289 (September or August 1977).

[Metcalfe1976]
Metcalfe, R. M. and Boggs, D. R., "Ethernet: Distributed Packet Switching for Local Computer Networks," *CACM* **19,7** pp. 395-404 (July 1976).

[Michel1976]
Michel, J. and vanDam, A., "Experience with Distributed Processing on a Host/Satellite Graphics System," *Computer Graphics* **10**(20)(1976).

[Mitchell1982]
Mitchell, J. G. and Dion, J., "A Comparison of Two Network-Based File Servers," *Communications of the ACM* **25**(4)(April 1982).

[Ni1981]
Ni, L. M. and Hwang, K., "Optimal Load Balancing Strategies for a Multiprocessor System," *Proceedings Tenth International Conference in Parallel Processing*, pp. 352-357 (August 1981).

[Oppen1983]
Oppen, D. C. and Dalal, Y. K., "The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment," *ACM Transactions on Office Information Systems* **1**(3) pp. 230-253 (July 1983).

[Popek1981]
Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G., and Thiel, G., "LOCUS: A Network Transparent, High Reliability Distributed System," *Proceedings Eight ACM Symposium on Operating System Principles*, pp. 169-177 (December 1981).

[Popek1984]
Popek, Jerry, *private communication*, LOCUS Corp. (1984).

[Postel1980b]
Postel, J., "DOD Standard Internet Protocol," RFC 760, Information Sciences Institute (January 1980).

[Postel1980a]
Postel, J., "DOD Standard Transmission Protocol," RFC 761, Information Sciences Institute (January 1980).

[Powell1983]
Powell, Michael L. and Miller, Barton P., "Process migration in DEMOS/MP," *Proceedings 9th ACM Symposium on Operating Systems Principles, Operating Systems Review* **17**(5) pp. 110-119 (October 1983).

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

[Rao1979]
Rao, G. S., Stone, H. S., and Hu, T. C., "Assignment of Tasks in a Distributed Processor System with Limited Memory," *IEEE Transactions on Computers* **c-28**(4)(April 1979).

[Ritchie and Thompson 78]
Ritchie, D. and Thompson, K., "UNIX Time-Sharing System," *Bell System Technical Journal* **57**(6) pp. 1905-1929 (1978).

[Rothnie1980]
Rothnie, J. B., Bernstein, P. A., Fox, S., Goodman, N., Hammer, M., Landers, A., Reeve, C., Shipman, D., and Wong, E., "Introduction to a System for Distributed Databases (SDD-1)," *ACM Trans. on Database Systems* **5**(1)(March 1980).

[Selinger1979]
Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G., "Access Path Selection in a Relational Database Management System," *Proceedings of ACM SIGMOD*, (1979).

[Selinger1980]
Selinger, P. G. and Adiba, M., "Access Path Selection in Distributed Database Management Systems," Research Report RJ2883 (36439) 8/1/80, IBM San Jose Research Lab (1980).

[Stankovic1981]
Stankovic, J. A., "The Analysis of a Decentralized Control Algorithm for Job Scheduling Utilizing Bayesian Decision Theory," *Proceedings Tenth International Conference in Parallel Processing*, pp. 333-340 (August 1981).

[Stankovic1983]
Stankovic, J. A., "A Heuristic for Cooperation among Decentralized Controllers," *Proceedings of INFOCOM*, pp. 331-339 (1983).

[Stone1977]
Stone, H. S., "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Transactions on Software Engineering* **SE-3**(1) pp. 83-93 (January 1977).

[Stone1978]
Stone, H. S., "Critical Load Factors in Two-Processor Distributed Systems," *IEEE Transactions on Software Engineering* **SE-4**(3) pp. 254-258 (May 1978).

[Stonebraker1976]
Stonebraker, M. R., Wong, E., Kreps, P., and Held, G. D., "Design and Implementation of INGRES," *ACM Trans. on Database Systems* **1**(3)(September 1976).

[Stonebraker1977]
Stonebraker, M. R., "A Distributed Version of INGRES," *Berkeley Workshop on Distributed Data Management*, Lawrence Berkeley Laboratory, (May 1977).

[Tantawi1984]
Tantawi, Asser N. and Towsley, Don, "Optimal Load Balancing in Distributed Computer Systems," Research Report RC10346 (#46163) 1/25/84, IBM Thomas J. Watson Research Center (1984).

[Williams1982]
Williams, R. et al, "R*: An Overview of the Architecture," *Proc. of The International Conf. on Data Bases*, (June 1982).

[Wong1976]
Wong, E. and Youssefi, K., "Decomposition - A Strategy for Query Processing," *ACM Trans. on Database Systems* **1**(3)(September 1976).

[Wu1980]
Wu, S. B. and Liu, M. T., "Assignment of Task and Resources for Distributed Processing," *Proceedings of Compcon80: Distributed Computing*, (1980).

[Youssefi1978]
Youssefi, K., "Query Processing for a Relational Database System," Ph.D. Dissertation, U.C. Berkeley, Electronics Research Laboratory, U.C. Berkeley (6 January 1978).

END