A Distributed Interactive Computer System

Tanenbaum, A.S.

1977

A Distributed Interactive Computer System

by

Andrew S. Tanenbaum

Vrije Universiteit

Amsterdam, The Netherlands

Abstract

A computer system in which each interactive user has his own dedicated CPU is described. These user machines are each connected to a central minicomputer on which the operating system runs. The user machines request operating system services such as file i/o by exchanging messages with the central machine. The proposed design is based on the idea of having the user machines interpret the machine language of a virtual machine whose object programs are very compact in order to reduce memory and bandwidth requirements. Various potential problems and bottlenecks are analyzed, including CPU performance, memory usage, and transmission time. We conclude that the minicomputer can probably provide over 100 user machines with operating system services.

1. INTRODUCTION

For some applications, interactive ("timesharing") use of computers is more convenient than non-interactive ("batch") use. Program development and debugging, computer aided instruction and many problems involving computer graphics fall into this category. In the old days a programmer could simply sign up for an hour of machine time and have the machine all to himself. As demand for time increased and machines became more expensive, timesharing systems were developed to make more efficient use of the machine than was possible using the previous "dedicated" mode of operation.

Unfortunately timesharing systems (and in fact all multiprogramming systems) have a severe drawback: they are exceedingly complicated. This complexity leads to high development costs (MULTICS cost more than $50 million [1]), high overhead both in terms of time and memory wasted, and systems that are unreliable and difficult to modify.

In this paper we discuss a new architecture for interactive computer systems that can greatly reduce this complexity, and thereby lead to simpler, more reliable systems. The basic idea is to give each user his own LSI microcomputer instead of having all the users share a single large CPU.

Rather than equipping each microcomputer with its own complement of expensive disks the system contains 1 or 2 central minicomputers to which the disks are attached. These minicomputers do NOT run user jobs, but merely provide file system services, message switching, and other typical operating system type services to the microcomputers upon request. The system described

here is currently under development by the Computer Science Group at the Vrije Universiteit.

2. OVERVIEW OF THE SYSTEM

Fig 1 gives a simplified picture of the system. Two configurations are being investigated: Unary Star, in Fig. 1a, and Binary Star, in Fig 1b. Unary Star has the advantage of relative simplicity, but the disadvantage of vulnerability. Binary Star is more complex, but also more failure tolerant. In each case the user has his own private machine (CPU, memory, and terminal), which we will refer to as the UM (User Machine). Each UM has a connection to the central node(s), which we will refer to as the OSM (Operating System Machine(s)).

All actual computing activity takes place in the UM's. When a person logs in, the job control interpreter residing in his UM asks for a command. A typical command would be to compile a program, in which case the UM requests the compiler and the source program from the OSM and loads them into the UM's memory. The compilation then occurs entirely within the UM.

A program in a UM can request service from the OSM by sending it a message. Typical requests are create a new file, open an existing file, read, write, give the time and date - in short - the same type of services provided by "normal" operating systems. When a read message is sent to the OSM, the resulting answer message from the OSM contains the data requested. Note that the only way a UM can get file, i/o, and other services performed is by sending a message to the OSM and waiting for the answer message. In the initial version, the types of services to

be provided by the OSM will be similar to those provided by UNIX[2].

This design differs from a distributed system using intelligent terminals in that not only is editing done locally, but also compilation, linking, and execution of all programs. Even number crunching is done in the UM's. The design differs from a traditional network in that the UM's are small monoprogrammed machines with no disks or peripherals whereas network nodes are typically larger multiprogrammed machines with a full set of peripherals. Furthermore, since a UM has neither a self contained operating system nor local files, it is not capable of running if isolated from the rest of the system, whereas a network can be characterized by the ability of its nodes, to run local jobs even if the network goes down. In other words, this design is neither for a single machine nor for a network, but rather for an intermediate form. Other distributed systems are described in [3-5].

In addition to the "personal computer" approach of the design presented thus far, we are also considering having a pool of microcomputers available for "background" jobs. If a user knew that a certain request e.g. a compilation, was going to take a substantial amount of time, he could request it to be performed on one of the pool processors, leaving his personal computer free for other activities during compilation.

A refinement of this approach would allow a job to be split over several pool processors, with the output of one being used as the input of the next. For example, one machine might read a program and perform the first pass of a compilation. The intermediate code output by pass 1 would be fed into a second pool

processor (as it was produced) for pass 2. Its output might be fed into yet a third machine for assembly. In UNIX terminology, these inter-machine connections are pipes, and the machines are functioning as filters. Of course the fact that the end of a pipe is on a different machine should be transparent to the programs using the pipe.

3. ADVANTAGES

The most important feature of this system is its extreme simplicity. Since none of the machines are multiprogrammed, nearly all of the complexity associated with scheduling, memory management, swapping etc. vanish. The operating system in the OSM can be very straightforward, since essentially all it does is accept, queue, and carry out requests to read and write files and messages.

A second unusual aspect is the ability to have different, and incompatible computers (UM's) use the same operating system. Since all communication between programs and the operating system is carried out by exchanging simple fixed format messages, computers from different manufacturers, with different instruction sets can be added to the system quite easily. This is in contrast to the present situation where each new computer must have its own, unique operating system. It is also in contrast to some experimental distributed and multiprocessor systems in which the choice of a particular microcomputer is deeply embedded in the design.

A third virtue is the relative ease by which the system capacity can be expanded in small increments. Up to a point one

can just add as many more UM's as desired. When the OSM finally saturates it will have to be upgraded or replaced, but none of the UM's will be affected.

A fourth point is that this system takes full advantage of the new LSI microprocessor technology with its attractive price/performance ratio without requiring any special hardware to be designed or built. Some of the experimental distributed machines being proposed are one of a kind systems that may or may not ever be produced commercially. This approach has the disadvantage that if no manufacturer decides to produce the machine, few end users will ever be able to take advantage of it, no matter how sophisticated it may be, simply because few users have the capability or inclination to do a major hardware design and construction job. A system such as ours can be assembled from a variety of existing microcomputers without a soldering iron. The UM-OSM connections are just 20 mA current loops.

A fifth consideration is the relative insensitivity to hardware failures. If the CPU or memory in one of the UM's fails, its demise only affects one user, and its repair does not necessitate interrupting service to other users. Of course, a failure in the OSM (in the Unary Star configuration) will affect all users, but statistically most CPU and memory failures will occur in the UM's since that is where most of them are. The Binary Star system will (hopefully) be relatively tolerant of any failures.

4. DISADVANTAGES

Once a substantial body of code exists for any computer

system, changing to a new computer is fraught with problems. Since LSI microcomputers are so cheap, and since new ones with yet greater speeds are being announced so frequently, when new machines are added to the system, the temptation to use the most recent ones (even if from another manufacturer) is much harder to resist than with large, expensive machines. Thus program portability from one microcomputer to another is really crucial to countering (psychological) obsolescence.

At present the cost of the UM's memory is much larger than the cost of its CPU. This is likely to continue for a number of years, at least. Unfortunately our design does not make efficient use of memory, since it cannot be shared among UM's. When a user is thinking his (UM's) memory is wasted. A method of reducing memory requirements is needed.

Since the UM's rely upon the OSM for file storage, when a program is executed, both the program and its input files and other data must be shipped across the communication line from the OSM to the UM. It is not difficult to imagine that the bandwidth of this line could become a bottleneck.

5. APPROACH TO OVERCOMING THE DISADVANTAGES

The portability problem can be attacked by providing each UM with an interpreter for a virtual machine. All software for the UM is to be written for the virtual machine, rather than for the UM's own machine language. In particular, compilers are to translate source programs into object programs for the virtual machine. These programs are then carried out by the interpreter running on the UM. The machine language will be specifically

designed for the purpose of allowing programs to be represented in as few bits as possible (to save memory and bandwidth) without sacrificing too much speed.

If all software is written in, or compiled into, the machine language of some carefully chosen virtual machine, then introducing a new LSI microcomputer into the system requires only writing one program for it - an interpreter for the virtual machine language. Once this is available, it will be able to run all the existing UM software. Of course, there is a penalty in execution time for using this approach but two points should be kept in mind: (1) the existence of large numbers of commercially available machines using this principle (e.g. all the IBM 370's) strongly suggests that the execution time penalty can be made acceptable and (2) a distributed timesharing system with tens or even hundreds of CPU's will have a huge amount of raw computing power, allowing execution time to be traded for other desirable characteristics. CPU performance will be discussed in detail later.

We have already developed a virtual machine language called EM-1 (Experimental Machine-1) to be interpreted by the UM's. It is specifically intended for use with block structured languages supporting recursion such as PASCAL, since systems programs such as compilers and editors are generally written in such languages nowadays.

EM-1 is stack oriented, with a variety of instructions for pushing and popping local variables, global variables (variables declared in the outermost block), intermediate level variables, constants, array elements, etc. The arithmetic and boolean

instructions fetch 1 or 2 operands from the stack and leave their result on the stack. Conditional jumps pop 1 or 2 values from the stack and jump forward if the specified condition is met. The chief virtue of a stack architecture is simplicity: it is easy to generate efficient code for it.

The EM-1 architecture also provides for a paged, segmented virtual memory. Each segment is 64K bytes. There are 256 instruction space and 256 data space segments. Procedures are called using (2 word) descriptors. Jumps may not change the current instruction segment, hence 16 bit addresses suffice for addressing the program itself. Most data references are to the stack, or to local variables, so only a few "address bits" are needed. Only for accessing array elements are 24 bit addresses needed; special array instructions are provided for these cases.

The most frequently occurring EM-1 instructions require only a single byte, which includes both the opcode and address. For example, PUSH LOCAL 0, PUSH LOCAL 1, PUSH LOCAL 2 etc. might be assigned opcodes 182, 183, 184, etc. By not having distinct "instruction" and "address" bits we eliminate decoding time and increase flexibility. The interpreter executes an instruction by fetching 1 byte and executing a 256 way branch (using a jump table) based on its value. A few opcodes are reserved for escapes to 2, 3, 4, 5, and 6 byte formats for less common instructions. Measurements show that the 250 most common opcode-address combinations account for about 85% of all opcode address combinations. This means that 85% of the instructions will be 1 byte long. Most of the rest will be 2 bytes long. The mean instruction length is about 1.3 bytes. Preliminary results [6] indicate that

this technique can compress programs by a factor of 3 compared to most existing third generation computers.

Once the price of interpretation has been paid, certain facilities can be added a a relatively low cost. These include not only the large virtual address space, but also overflow checking on arithmetic instructions, checks for uninitialized variables (-32768 is used), and detailed performance profiles.

6. CPU PERFORMANCE

In the next 4 sections we examine the expected performance of the distributed system and attempt to analyze the most likely bottlenecks.

The speed of the interpreted EM-1 machine is determined largely by the number of host machine instructions required to carry out the most common EM-1 instructions pushing and popping values from the stack, and short conditional branches forward, since these are the ingredients of assignment, IF, and WHILE statements. We assume the host machine has a facility for manipulating 8 bit bytes and 16 bit words, and has a few 16 bit registers. The main loop of the interpreter must perform the following operations:

1. Fetch the next instruction byte into a register.
2. Increment the program counter.
3. Execute a 256 way branch (e.g. indexed jump) using the instruction as the index.

The exact timing will depend on the host machine's architecture and speed, but several existing 16 bit LSI microprocessors can perform these 3 functions in about 5-8 microseconds.

Each of the 256 1-byte opcodes must have its own execution routine but in many cases this will only be 2 or 3 instructions. Pushing local variable 3, for example, requires simply fetching the value, pushing it on the stack and jumping back to the main loop. Note that no bit fields need be extracted since the 256 way branch jumps to different locations for PUSH LOCAL 3, PUSH LOCAL 4, etc. It seems reasonable to assume the EM-1 machine will have an instruction time of about 10-12 microseconds. If the LSI microprocessor is microprogrammable, the EM-1 instructions time can probably be reduced to 5 microseconds.

The impact of the virtual memory is kept reasonable by a clever trick. The program counter, PC, is stored within the interpreter as a physical, rather than a virtual address. The address of the last word on the current code page, LIMIT, is kept in a register. Before each instruction byte is fetched, a test is made to see if PC <= LIMIT. If so, no action is needed; if not, the true virtual address must be computed, and PC and LIMIT updated. The important thing to note is that most of the time no action will be needed (most instructions are on the same page as their predecessor), so the only overhead is a register-register compare and conditional branch. Jumps and procedure calls must be handled specially, but if jumps are all pc relative, no special action is needed for any forward jump, no matter how far (if it jumps off page, the PC <= LIMIT test will detect that fact before the next instruction fetch). We estimate that the cost of implementing a large paged, segmented, virtual address space without any hardware assistance will be about 20-30% of performance, putting instruction execution times in the range 12-15 microseconds.

Barely 15 years ago there were many universities and scientific computer centers that possessed a single IBM 709 (cycle time = 12 microseconds) which served the entire user population. In our design each user has his own, private, machine comparable to a 709. Preliminary measurements show that a PASCAL compiler running on EM-1 should be able to compile 15 lines/sec, or a page of source text in 4 seconds of CPU time.

It is also worth noting that the stack architecture of EM-1 allows simple compilers to produce locally optimal code, whereas compilers for traditional multiregister machines often lose a factor of 3 or 4 in speed [7]. A 12 microsecond machine whose programs are near optimal is only a factor of 4 worse than a 1 microsecond machine whose programs are slowed by a factor of 3 due to bad code. The conclusion we draw is that the performance loss due to interpretation will be acceptable.

If this should subsequently prove to be untrue, we can always adopt a mixed strategy of interpretation and compilation. The most heavily executed portions of the key programs can be compiled instead of interpreted. By varying the amount of code compiled, we can trade time against space. Most programs have the property that a small part of the code is responsible for most of the execution time, so this strategy is an attractive one; a small increase in size may produce a large performance gain.

Raw CPU speed is only part of the story. When comparing two UM's, one using this interpretation scheme and one not, one must consider what would happen if the non-interpreted program did not fit in memory. It would have to be written as a collection of

overlays, or software paging would have to be employed. In both cases the resulting i/o activity would greatly degrade performance, making interpreation a far more attractive strategy than it might at first appear to be.

7. MEMORY REQUIREMENTS

One way to estimate how much memory each UM should have is to ask how much is needed to compile large programs. The preliminary version of the PASCAL compiler can compile itself in 48K bytes, used as follows: 24K bytes for the compiler itself, 16K bytes for the stack and workspace, and 8K bytes for the interpreter and its data. The interpreter is permanently resident in the UM and need only be loaded when the system is dead started. The 24K bytes for the compiler includes the entire compiler; when it is modified to take advantage of the EM-1 virtual memory, less space will be needed since it will not be necessary for the entire compiler to be resident at once.

Amdahl's rule states that a computer needs 1 megabyte of memory per MIPS. With an instruction time of 12 microseconds, (0.083 MIPS) this rule indicates that 83K bytes of memory are needed. However, EM-1 programs are a factor of 3 smaller than their IBM 360 equivalents, on which this rule is based. Data (e.g. source programs) cannot be compressed as easily and still be convenient to use, but by using the 128 unúsed character codes available in an 8 bit byte to encode common strings (e.g. BEGIN, END, THEN, ELSE, :=) some compression may be possible. If an overall factor of 2 is achieved, we need 40-45K bytes of memory.

Since most byte addressable microcomputers have an addressing limit of 16 bits (i.e. 64K bytes) we estimate that each UM

should have 48K-64K bytes of memory.

8. TRANSMISSION BANDWIDTH

The connection between the UM's and the OSM could be accomplished by a single dedicated wire, several dedicated wires in parallel, a message switching network, a packet switching network, a broadcast network, etc. The simplest and cheapest way is to use a single 20mA current loop interface. However, since 9600 baud is usually the highest bandwidth available off-the-shelf, we consider using n lines in parallel, i.e. the first 1/n bytes are sent over line 0, the next 1/n bytes over line 1, etc. For n=4, the asynchronous transmission (1 start bit and 1 stop bit per byte) rate is 4*960 = 3840 bytes/sec for Unary Star. For Binary Star this might mean 1920 bytes/sec. to each of the central nodes.

Since the response time to edit commands will be essentially instantaneous, independent of system load, (once the editor and data file have been loaded into the UM) let us examine compilation, where the transmission time may be more of a problem. Measurements show that an r line PASCAL program contains 32r characters, requires r/15 seconds of EM-1 CPU time for compilation, and generates 38r characters of assembly code output.

Let us assume that due to overhead, queueing delays for the disks, etc., an effective bandwidth of 3000 bytes/sec can be achieved. This means that each page of source text requires 640 ms to be read in, 4000 ms to be compiled, and 760 ms to be written out. If the i/o can proceed concurrently with the processing, the transmission time is essentially free.

The time required to load the 24K byte compiler is 8 seconds. Thus we arrive at the following rough estimates for compilation time:

Program Size	No i/o overlap	Overlap
1 page	13 sec.	12 sec.
5 pages	35 sec.	28 sec.
10 pages	62 sec.	48 sec.
50 pages	278 sec.	208 sec.

It should be noted that these are real times and should be compared to the response times of conventional timesharing systems, not the CPU time used. Note that separate compilation of procedures becomes an attractive technique under these circumstances.

9. MAXIMUM NUMBER OF USERS

In this section we examine the factors that limit the number of UM's that can be attached to one OSM, and try to estimate the maximum number of users that can be given good service. For simplicity we consider the Unary Star configuration with one disk. The OSM does little except process requests for reading and writing files, an activity that requires very little CPU time. However, three other potentially limiting factors are present: memory cycles, disk accesses, and buffer space.

Minicomputer i/o generally works on a cycle stealing basis. This limits the number of simultaneous transmissions, since the interfaces to the UM's cannot steal more cycles than there are. For reading, each buffer in the OSM's memory must first be filled

from the disk and then emptied by the communication interfaces. Allocating half the cycles to the UM's, a quarter to the disk (minicomputer disks transfer 2 bytes/cycle, serial interfaces 1 byte/cycle) and a quarter to the CPU and low speed i/o and assuming a 1 microsecond memory cycle, there are enough cycles available for 500000/960n simultaneous transmissions (n is the number of 9600 baud asynchronous serial interfaces running in parallel per UM). For n=4, this is 130 UM's.

The maximum number of simultaneous transmissions is potentially limited by the number of disk accesses per second. Most UM requests will require one disk read or write. We assume that disk seeks are largely overlapped, yielding an average effective seek time of s ms, that rotational latency is 8.3 ms, and that an average transfer takes t ms. Under these conditions, 1000/(s+8.3+t) requests/sec can be satisfied. High performance minicomputer disks transfer at a rate in excess of 1Mb/sec. For mean block sizes between 1K and 4K bytes, (typical of page sizes in virtual memory systems) about 25 requests/sec be satisfied if s=30, ms for the non-overlapped portion of the average seek.

In order to handle 25 requests/sec on a sustained basis, sufficient buffers are needed in the OSM memory. For request sizes of 1K to 4K bytes, 6K to 100K bytes of buffer space are required. Depending on the distribution of request sizes and the amount of OSM memory available, lack of buffer space could limit the number of simultaneous transmissions.

Based on this data, the limiting factor appears to be access to the disk. The number of users that can be granted a given level of service depends upon the rate at which they generate

requests. If requests come in too fast, a queue builds up and service degrades. To make a rough estimate of the maximum number of active users, we use an M/M/1 finite population queueing model to estimate the number of UM's required to saturate the disk. We define the disk to be saturated when a request must wait in the queue for a time equal to its service (transmission) time.

We have used the M/M/1 queueing model to derive the maximum number of UM's, but we prefer to give a more intuitive argument here. Let Q be the mean number of UM's with service requests either queued or being processed. These UM's generate no input. If is the input rate of a single UM, the effective request rate to the OSM from the N-Q unblocked users is (N-Q) . The disk system can process = 25 requests/sec.

In equilibrium in a saturated system (i.e. disk utilization = 1), we have (N-Q) = . This relation is maintained by the negative feedback nature of the system. If the disk service improves it reduces Q, which increases the rate at which work arrives, which in turn increases Q again. If the disk slows down, Q builds up and there is less input, which allows the disk to catch up and reduce Q. From our definition of saturated, Q=2 approximately, since a wait time equal to service time implies 1 customer waiting and 1 being served. Solving for N gives: N = 2+

Now we will estimate the data rate for a user engaged in the edit, compile, assemble, execute, crash cycle known as debugging. Based on preliminary measurements, we project the combined sizes of the editor, compiler and assembler at 40K bytes. The editor reads in the source text, and writes it out later. The compiler reads it back in, and writes out assembly code, which is read

back in by the assembler. Finally the binary code is written out and read back in for execution.

Assuming a page of source text is 60 x 32 = 1920 bytes, the assembly code generated by it is 60 x 38 = 2280 bytes, and the object code is 60 x 8 bytes, each page of source text generates 11280 bytes woth of data transmission. The total data transmitted per cycle is 40000+11280*P bytes, where P is the program size in pages. If data is requested in blocks of B bytes, and the debug cycle takes T seconds, the number of requests per second is

$$= (40000 + 11280 * P) / (B*T)$$

Given this, we can compute the number of users required to cause disk queueing to become noticeable. A few typical values for a 10 minute cycle are shown below for 1,5, and 20 page programs, and block sizes of 512, 1024, 2048, and 4096 bytes.

	1 page	5 pages	20 pages
512	152	81	30
1024	302	160	59
2048	601	318	116
4096	1200	635	229

10. SIMULATION RESULTS

In this section the results of a simulation study will be presented. The model is again of Unary Star, with a single disk (using the elevator algorithm). Finite buffer space, queueing, and transmission delays are included in the model.

Choosing a performance metric is more difficult than with

conventional timesharing systems. Command response time is not reasonable since the response time to edit commands is essentially zero, independent of the system load, and one does not usually speak of the "response time" in connection with compilations and other long requests.

To determine how many users can be supported before requests for data begin to bog down, we again define saturation as the point where the response to a single request for data (e.g. read 512 bytes) takes twice as long as it would in an empty system. Figs. 2 and 3 show this response time as a function of the number of users for various combinations of block size and program size (10 minute debug cycle). Two things are obvious: if the users are working with large programs, the system saturates faster than if they are working with small programs (hardly surprising); making a smaller number of large requests is preferable to making a larger number of small requests (because the disk access time is important, and fewer requests means fewer seeks), except where lack of buffer space begins to become important. The maximum number of users is indicated.

11. CONCLUSION

A computer system consisting of many microcomputers attached to a central minicomputer is proposed as an alternative to a large centralized timesharing system. To keep memory and transmission requirements down, the microcomputers interpret a machine language designed for minimizing object program size. A rough calculation and simulation indicate than one minicomputer can can provide operating system services for over 100 users.

REFERENCES

[1] R. Graham, Use of High Level Languages for System Programming, Project Mac Report TM-13, MIT, Sept. 1970.

[2] D.M. Ritchie and K. Thompson, The UNIX Timesharing System, CACM, vol. 17, no. 7, July 1974, 365-375.

[3] W.A. Wulf and C.G. Bell, C.mmp A Multi-mini-processor, FJCC, vol 41, 1972, 765-777.

[4] S. Davidson, A Network of Dynamically Microprogrammable Machines, IEEE 8th Annual Workshop on Microprogramming Supplement.

[5] B. Arden and A. Berenbaum, A Multi-microprocessor Computer System Architecture, Operating System Review, vol. 9, no. 5, 1975, 114-121.

[6] A. Tanenbaum, Implications of Structured Programming for Machine Architecture, to be published.

[7] A. Tanenbaum, A General Purpose Macro Processor as a Poor Man's Compiler-Compiler,IEEE Trans. on Soft. Eng., vol. SE-2, no. 2, June 1976, 121-125.

[8] A. Scherr, An Analysis of Time-Shared Computer Systems, MIT Press, Cambridge, Mass., 1967.

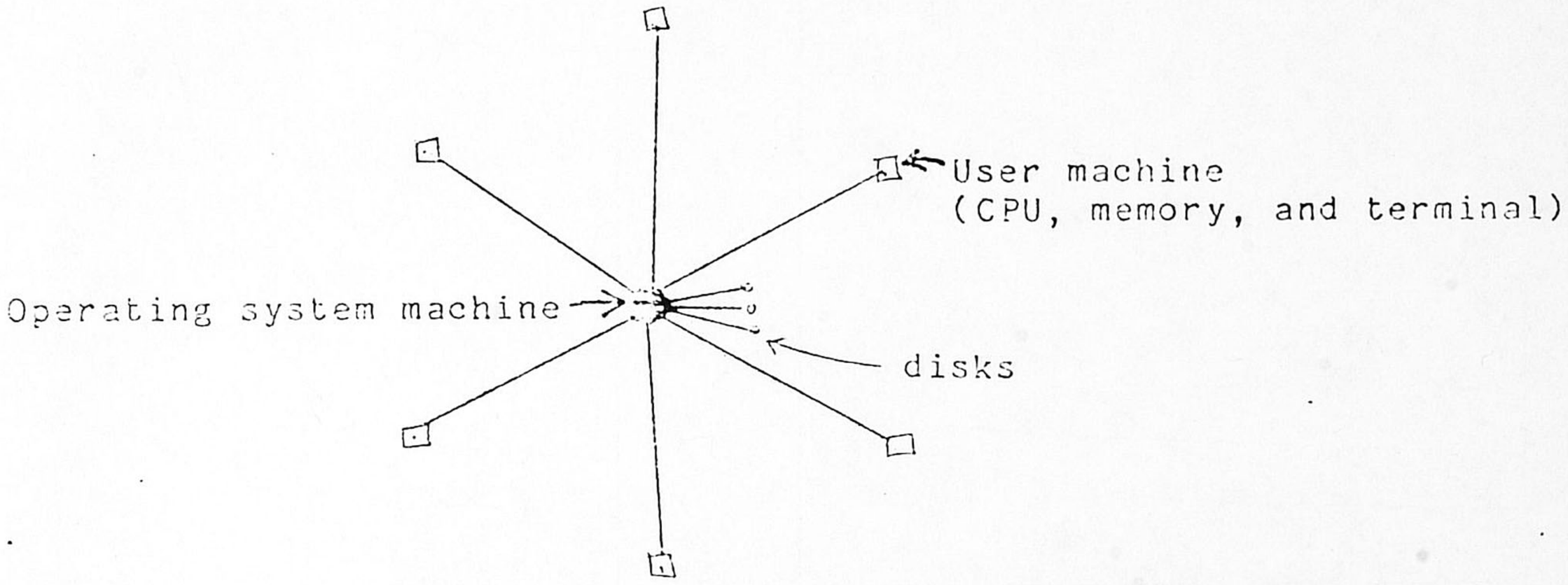


Fig 1a Unary Star

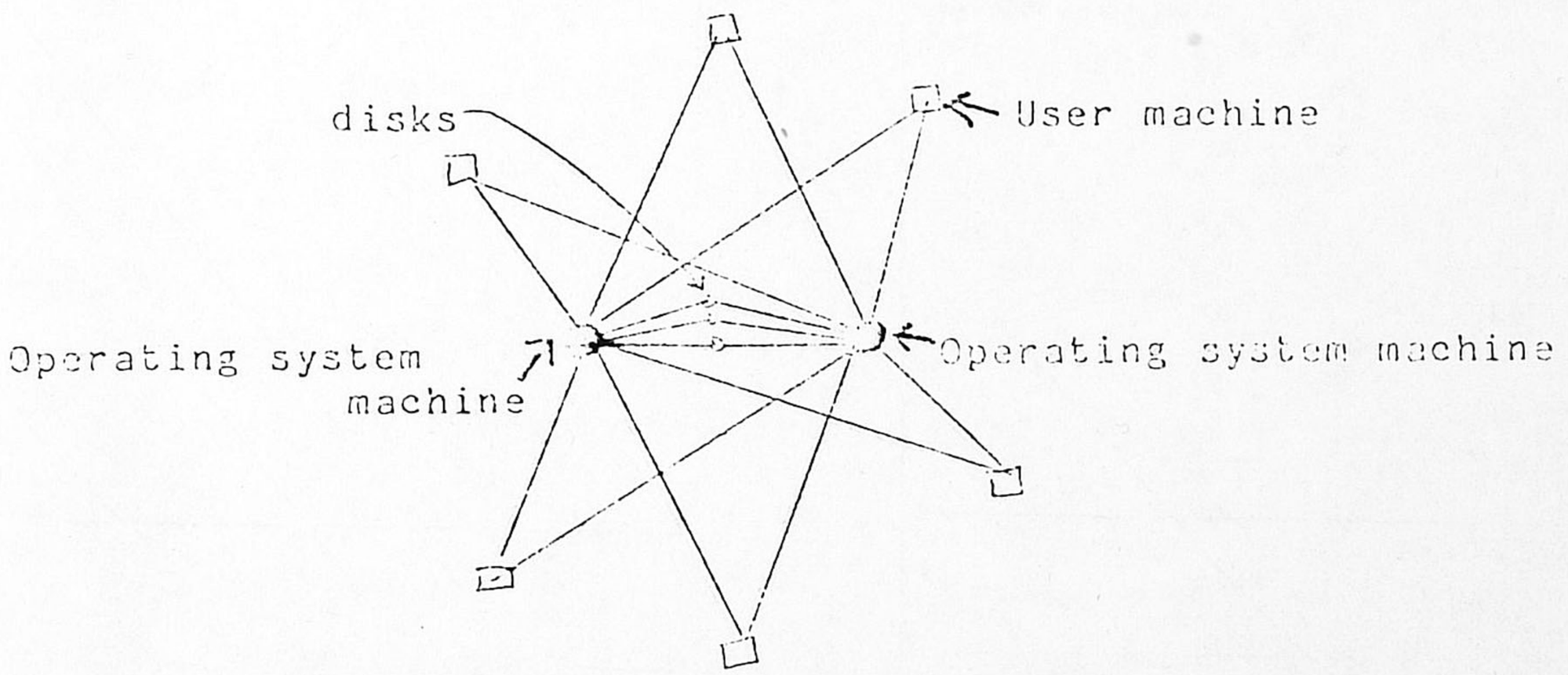
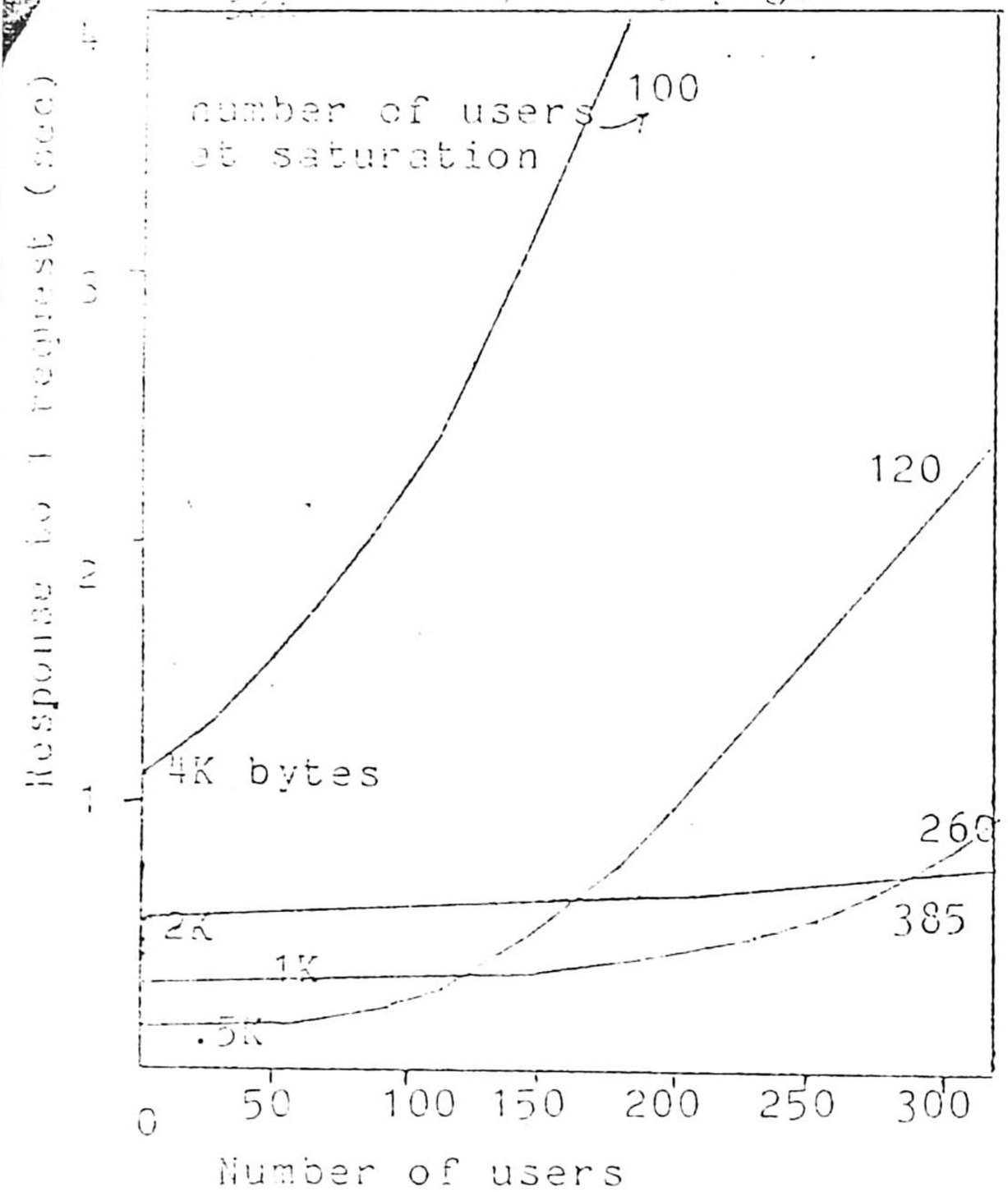
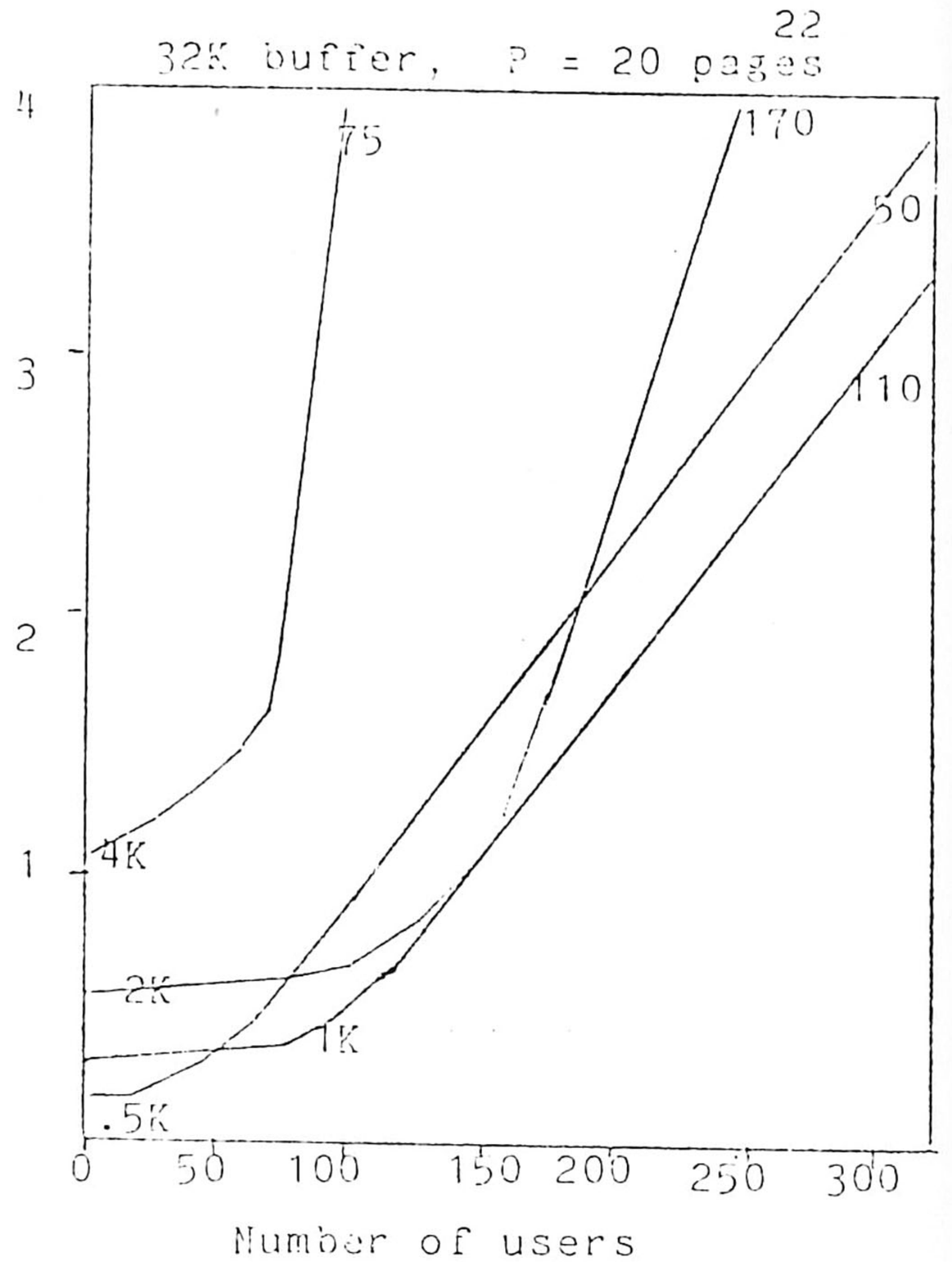
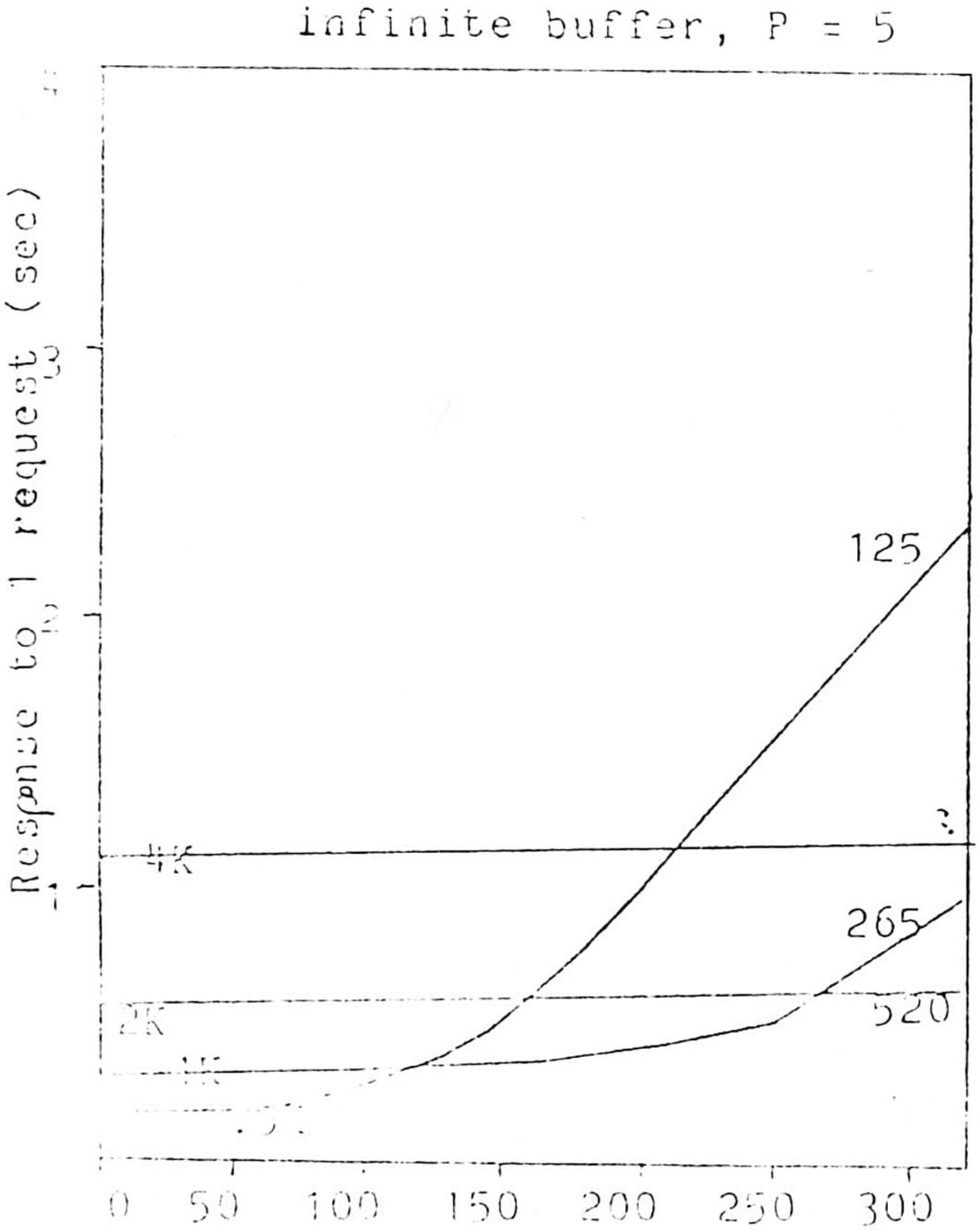


Fig 1b Binary Star

B022583-2







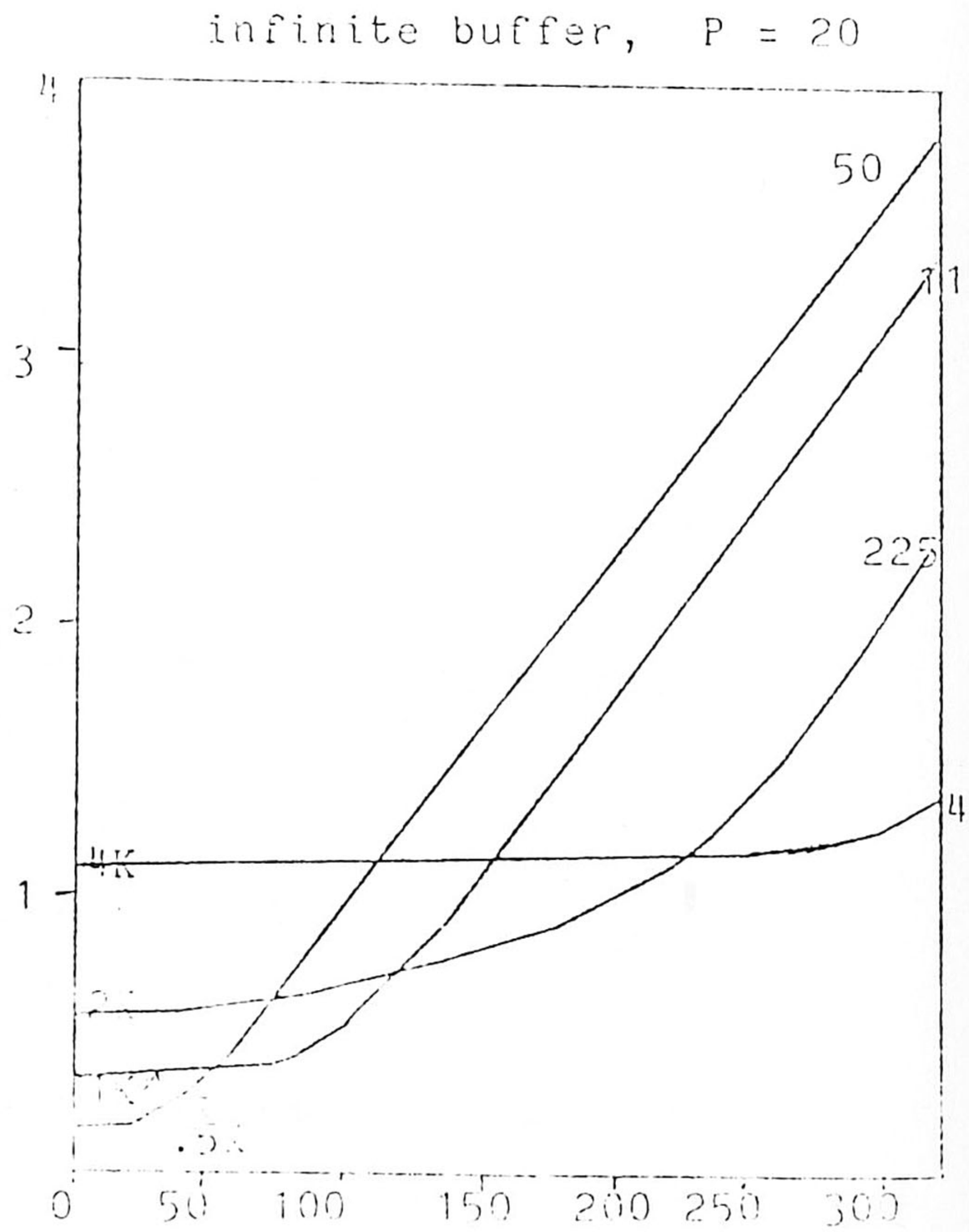


Fig. 2