PLACEMENT OF PROCESSES AND FILES IN DISTRIBUTED SYSTEMS

Kriton Kyrimis

CS-TR-250-90

June 1990

(Thesis)

PLACEMENT OF PROCESSES AND FILES IN DISTRIBUTED SYSTEMS

Kriton Kyrimis

A DISSERTATION
PRESENTED TO THE
FACULTY OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE BY THE
DEPARTMENT OF
COMPUTER SCIENCE

JUNE 1990

To my advisor, Rafael Alonso

Acknowledgments

I would like to thank my advisor, Professor Rafael Alonso, who directed this research and to whom this thesis is dedicated. I would also like to thank the readers of my thesis for their helpful comments, Professor Hector Garcia-Molina and Professor David Hanson, who took time off his sabbatical to help bring my writing style to an acceptable level.

A special thanks must go to the members of the CS staff, past and present, who maintain our computing environment. Their work is always appreciated even though we, as users of this environment, sometimes fail to express this appreciation.

Finally, I would like to thank the office staff of the Department of Computer Science and, in particular, Sharon Rodgers, who went out of their way to make a graduate student's life bearable.

This work was supported by an IBM Research Initiation Grant and a grant from DRI Sarnoff laboratory.

Abstract

In this thesis, we examine ways of improving the average job performance in a distributed system by controlling on which machine processes or files are placed. We describe an implementation of process migration under Sun UNIX, and use this implementation in experiments to assess the relative merits of process migration and initial placement as load-balancing strategies. Finally, we examine dynamic file caching in distributed systems where maintaining file consistency is important. Using an analytical model, we derive a general caching algorithm. In a series of simulation studies, we compare the performance of this algorithm with that of the simpler strategies of never caching and always caching files for various network configurations.

Table of Contents

CHAPTER 1

Introduction

In a distributed computing environment, the average response time of a job can be improved by using all of the machines in the local network instead of just the single machine in a mainframe environment. Response time is the interval between the time a job starts and the time it completes its execution.

Placement of processes and resources can be controlled explicitly in a distributed system. Explicit placement can improve job performance by insuring that a job runs at its most suitable site, e.g., a fast CPU, or one with a small load, or near the resources that the job uses. Placing a job on the machine where its files are stored avoids costly network communications, for example.

In a multi-machine network, the load on each machine varies from light to heavy. In this imbalanced situation, users of highly loaded machines suffer because they cannot take advantage of the computing power available from the lightly loaded machines. A *load balancing* mechanism places processes on machines so that the average response time is minimized. In this balanced situation, the average response time for all users is improved.

Load balancing is either non-preemptive or preemptive. A non-preemptive strategy decides on which machine a process will run when it is initiated. Once placed, a process remains on a specific machine. A preemptive strategy is just the opposite. A process is initiated on one machine and possibly moved to other machines by the system. Such movements occur in order to balance the load.

A preemptive strategy requires operating system support for *process migration*, i.e., moving processes among machines. Few existing systems provide such support. The first part of this thesis describes an implementation of a process migration facility in the Sun UNIX operating system. Having a support for process migration permits experimental investigation of the relative merits of the two approaches to load balancing. Chapter 3 details the outcome of such experiments.

The average response time of a process is reduced if its resources are available locally. For instance, file I/O accounts for a significant fraction of the execution time in many applications. Therefore, placing files locally reduces I/O costs by avoiding network costs, and the second part of this thesis focuses on file placement.

File placement is either static or dynamic. In static placement, files are placed at the machine on which the process will run, which is assumed to be known. This strategy is called initial file placement, and optimum initial file placement has been studied thoroughly. Optimum placement is known to be NP-complete, and various heuristics have been proposed. (See Reference 1 for a survey.)

In dynamic placement, files are placed on and perhaps subsequently moved to machines so that the average response time of all jobs is minimized. No assumptions about process placement are made in this strategy. File caching is a variation of dynamic placement in which files are not moved, but are copied to other machines in order to improve access time. These copies are removed when they are no longer needed. While file caching has been implemented in several systems, most deal poorly with maintaining consistency. Chapter 4 gives examples of such systems and describes a simple analytical model and the resulting caching algorithm that maintains file consistency without sacrificing performance.

CHAPTER 2

A Process Migration Implementation on SUN Operating System

Process migration is a useful tool for applications such as load balancing and process checkpointing. Users can also use this facility to move a process from a machine that is about to go down. The interface to the process migration mechanism must be transparent to the processes that are being migrated, i.e., processes must be unaware of the process migration mechanism. After a process is moved, it must continue execution as if it were still running on the original machine. The performance of this mechanism may vary depending on the purpose for which it is used. If process migration is used for load balancing, it must have little overhead. It must be fast, requiring time comparable to the time it takes to initiate a program. If process migration is used for moving individual, CPU intensive processes in order to improve their performance, or to remove important tasks from a machine that is about to be halted, less efficient mechanisms are acceptable.

Implementing process migration is non-trivial. One must keep track of all the system resources that a process is using, and to reallocate them to the migrated process on the destination machine. For example, a migrated process must be able to access the files that it was accessing before the migration from the destination machine. Moreover, the system must maintain all interprocess communication connections to the migrated process. Providing all these capabilities can be very hard, if not impossible.

As described in Section 2.2, process migration has been implemented in a few operating systems. However, all these systems have been designed specifically as distributed systems. On a UNIX-based system, like that described in Section 2.3, process migration is unavailable. The remainder of this Chapter describes a process migration mechanism for such a UNIX system. Section 2.4 describes the user interface at both the command line and the programming levels. Section 2.5 describes the minor modifications made to the UNIX kernel to support process migration. Section 2.6 gives a performance evaluation of the modified kernel and the associated process migration commands. Section 2.7 discusses the limitations of the implementation. Despite these limitations, the implementation is successful as demonstrated by the tools for manual load balancing described in Section 2.8.

2.2. Other implementations

There are only four other implementations of process migration of which we were aware at the time we built our system. Since that time, there have been several other implementations; in the 1988 Winter Usenix Conference where an earlier version of this work was presented [2], a session was devoted to process migration implementations.

In the DEMOS/MP operating system [3] all interaction with the kernel is via message passing. This mode extends to the kernel itself, which uses message passing to communicate with kernels of other machines. To migrate a process, one kernel transfers the state of a process to another kernel. After the process is created on the other machine, all pending messages are forwarded to the new address of the process. Finally, the old process is destroyed and replaced with a process that acts as a forwarding agent, which ensures that subsequent messages to the migrated process are delivered correctly.

The Locus distributed UNIX system [4] attempts to distribute all the resources of a program, including the CPU, among all the machines in a network, in order to achieve network transparency for all resources. The `migrate` system call changes the execution site of a process.

The V-System [5] also provides a network transparent environment. Each machine runs a functionally identical copy of a distributed kernel that provides address spaces, processes that run in these address spaces, and interprocess communication. Address spaces are grouped into logical hosts, and processes are bound to a logical host, which allows processes to have a unique, global address. The `migrateprog` command migrates a process; it copies the state of a process to the destination machine and then repeatedly copies that part of the state that has changed since the previous copy, until relatively little information is copied. At this point, the old process is frozen and remaining modifications in its state are copied to the new machine. This pre-copying reduces the time that a process remains frozen thus increasing performance. The old process is destroyed and the new process is bound to the logical host of the old process so that the new process is indistinguishable from the old. Finally, the new process continues. While the process is frozen, a kernel server that is executing inside the kernel sends ''reply pending'' packets to all processes that have sent messages to the process being migrated so that these processes do not time out.

Finally, in the Sprite operating system [6, 7], when a process is migrated, system calls that can have different effects if executed on different machines (e.g., get time of day, get process id) are executed on the original machine. Remote execution of system calls is accomplished by exchanging messages between the process and the kernel of the original machine. In this way, although a process may be physically located on a different machine, it continues to execute, in part, under the kernel with which it was initiated.

All of these implementations have relied on special features of the operating system that create a distributed environment and make it relatively easy for two machines to cooperate in migrating processes. However, ''ordinary'' operating systems such as Sun O.S., a UNIX implementation that evolved to its current state but was not designed for distributed use, do not have such features. The lack of these features makes it difficult to implement process migration in these kinds of systems.

UNIX does not support communication between two kernels, so our process migration implementation is limited; not all processes can be migrated. Apart from badly behaved processes (which are discussed in Section 2.7), the main limitation is that processes that use pipes and sockets, which are inter-process communication facilities, cannot be migrated.

2.3. Implementation environment

Our process migration system is intended for a homogeneous network; processes can migrate only among machines of the same architecture. The implementation was made under version 3.3 of Sun O.S., which is a derivative of the Berkeley 4.2 BSD version of the UNIX operating system.

The machines and a file server were connected by a 10 Mbit Ethernet. Each workstation's local files, along with the files on the file server, are available on every machine via Sun's Network Filesystem (NFS) [8, 9]. NFS permits a file system to be accessed as a part of the directory structure of a local machine. On our system, the root directory of each machine is ''mounted'' on the `n` subdirectory of the root directory of all other machines. For example, the root directory of a machine called `brador` can be accessed from other machines as `/n/brador`.

2.4. User-level description

Our process migration system can be used at both the command and at the programming levels. At the command level, the system provides three new commands for migrating processes. At the programming level the system provides a new signal† and a new system call. Naïve users can use the new commands to migrate processes without having to write code. Knowledgeable users can use either the new commands or the programming interface directly in their own programs.

2.4.1. User commands

Most of the process migration code is user-level code, i.e., the process migration commands are user applications. However, if the commands do not satisfy specific needs, users can write substitutes (see Section 2.4.2).

The process migration system provides three commands, which cover most of the common cases: `dumpproc`, `restart` and `migrate`.

`dumpproc` terminates a process and writes all the information that is necessary to restart it to disk. The process is specified by giving its process id as a command argument. For security reasons, only the superuser or the owner of the process can kill a process.

`restart` restarts a process that was killed on some machine with `dumpproc`. Command options specify the process id of the process, and the name of the machine on which the process was dumped; the default machine is the current machine. The process is restarted on the machine on which the command was given and at the terminal on

† Signals in UNIX are essentially software interrupts; see References 10, 11

which the command was typed. All files that were open when the process was dumped are available to the restarted process with the correct access modes and offsets. Of course, if these files have been modified in the interim, the result of continuing the process will be unpredictable. Terminal modes such as ''raw'' (process input characters as soon as they are typed) or ''noecho'' (disable echoing of input characters) are preserved, so that visual applications like screen editors can be restarted properly. For security reasons, only the superuser or the owner of the original process can restart a process.

`migrate` moves a process from one machine to another. `migrate` is a combination of `dumpproc` and `restart`, which avoids having to go to another terminal to issue `dumpproc` or `restart`. Command options specify the process id of the process to be migrated, the name of the machine from which the process is to be moved, and the name of the destination machine; the default for both the source and the destination machine is the current machine. The migrated process is restarted on the terminal where `migrate` was typed, which need not be connected to the machine where the migrated process will run. `migrate` calls `dumpproc` and `restart` internally, using the remote execution command `on` [12] of the Sun 3.3 operating system, if necessary.

As an example of a typical use of the process migration mechanism, suppose that we are running a program on a machine called `brick` and that we wish to migrate it to a machine called `schooner`. First, we determine its process id using the UNIX `ps` [12] command; assume that the process id is 1234. We can migrate the process with `dumpproc` and `restart`:

```
    brick: dumpproc -p 1234
    schooner: restart -p 1234 -h brick
```

The prompts `brick:` and `schooner:` identify the machine on which the commands shown are typed. Alternatively, we can issue

```
    migrate -p 1234 -f brick -t schooner
```

on *any* machine.

2.4.2. Programming interface

Process migration is supported by a new signal and a new system call. These can be used to create programs that handle process migration in arbitrary ways.

When a process receives the new signal, `SIGDUMP`, which is sent by the `kill` system call, the process is terminated and all the information that is necessary to restart it is written to three files in a standard directory (e.g., `/tmp` on the file sever). The three files are named `a.outXXXXX`, `filesXXXXX` and `stackXXXXX`, where `XXXXX` is the process id of the terminated process. Each file name is prefixed by the name of the machine on which the original process was running to distinguish between dumps from different machines. For example, the full name of the first of these files, created by dumping process 1234 on machine `brick` is `/tmp/brick.a.out1234`.

The first file is an executable image, created by dumping a header and the text and data segments of the process. If this file is executed, the execution will be similar to running the original process from its beginning, except that all static variables will be initialized to the values that they had when the process was killed. To resume execution from the point where the process was terminated, the new system call `rest_proc`, described below, must be used instead. Having the ability to create such a file obviates the need for the `undump` utility, which combines an executable and a standard core dump, creating a new executable. `undump` is used to build programs with large initialized data structures, thus minimizing startup time. The stack segment of the process is written to the `stackXXXXX` file, described below. Processes that use `SIGDUMP` to initialize their data structures should check to see if these data structures have been initialized or not, and proceed accordingly.

The second file contains the information that is not needed by the kernel to restart the process. These data are used at the user level when the process is restarted. The contents of this file are as follows.

– A ‘‘magic number’’ (octal 445) that identifies this file as a `filesXXXXX` file.

– The name of the machine on which the process was running when it was killed.

– The absolute path name of the current working directory.

– For each entry in the open file table of the process (which has a fixed size), an indicator specifying whether the entry refers to an open pipe/socket, an open file, or it is unused. For open files, this indicator is followed by the absolute path name of the file, the file access flags (e.g., read only etc.), and the file offset. No information is kept for pipes and sockets, because the process migration mechanism cannot migrate processes that use them.

– The terminal flags, (e.g., raw mode, echo/noecho, etc.)

The modified kernel can produce all the paths mentioned above because it keeps copies of the name of the current working directory and of all open files. The copy of the name of the current working directory is updated whenever the current working directory is changed. Each time a file is created, opened, or closed, the name of the file is combined with the current working directory to produce the absolute path of the file. These paths are constructed using names provided by user processes to the kernel, so paths with symbolic links†, are not resolved in the `filesXXXXX` file. Not resolving symbolic links may cause problems when trying to reopen a file after the process is restarted. Consider, for example, `/usr/foo`, on a machine called `classic`. If `/usr` is a symbolic link to `/n/brador/usr`, then `/usr/foo` is actually `/n/brador/usr/foo`. Suppose a process opens `/usr/foo` and is then terminated with `SIGDUMP`. When the process is restarted, `/usr/foo` must be reopened. One way is to prepend `/n/classic` to the old name and open `/n/classic/usr/foo`, which, because of the symbolic link, is actually `/n/classic/n/brador/usr/foo`. Unfortunately, NFS forbids this syntax, so using this file name would not produce the desired result.

This problem can be solved by processing `filesXXXXX` before it is used, resolving the names of any symbolic links contained therein. The Sun 3.3 operating system `readlink` [13] system call can be used iteratively to resolve all symbolic links in a path. `dumpproc` uses `readlink` to postprocess `filesXXXXX` on behalf of the user; the file names stored in a `filesXXXXX` file by *dumpproc* can be used without

† Symbolic links are files containing the name of another file, so that the latter can be accessed by a different name. For example, on our system, a user's home directory, `/u/user`, is actually a symbolic link to a directory on the file server, e.g., `/n/brador/u2/user`.

modification.

The third file contains the information that is needed by the kernel to restart a process. The contents of this file are as follows.

– A ''magic number'' (octal 444) that identifies this file as a `stackXXXXX` file.

– The user credentials (e.g., user and group id).

– The size and contents of the stack.

– The contents of all the registers.

– All the information kept in the user and process structures that is related to the disposition of signals (e.g., which signals are being caught or ignored, which functions are handling those signals that are caught, etc.)

The information contained in these files is used by the kernel and by the process that restarts the terminated process as described below.

A new system call, `rest_proc`, restarts a process that was terminated using `SIGDUMP`. It takes two arguments, the names of the `a.outXXXXX` and `stackXXXXX` files mentioned above; the information in the `filesXXXXX` file is used by the user-level program that invokes `rest_proc`. The effect of `rest_proc` is to overlay the current process with a copy of the process from which the two files are created, resuming execution from the point where the process was killed. Normally, there is no return from `rest_proc`; it returns only if the system has insufficient resources to create the new process, or if something was wrong with the two files, e.g., if they did not exist or if their format was incorrect. `rest_proc` is similar to the UNIX `execve` [13] system call, which is used with the `fork` [13] system call to create new processes.

Before issuing `rest_proc` to restart a process, a program should perform the following actions.

1. Read the `filesXXXXX` file.

2. Set the real and effective user id to the ones read from `filesXXXXX`, using the `setreuid` [13] system call.

3. Change the current working directory to the one read from `filesXXXXX`, using the `chdir` [13] system call.

4. Open all files that were open when the old process was running, assigning the same file descriptors. These files must be opened with the correct access modes and positioned at the correct offset using the information stored in `filesXXXXX`.

2.4.3. Example–dumpproc and restart

`dumpproc` uses the `SIGDUMP` signal to terminate a process. It performs the following steps.

1. Creates a lock file (i.e. one that cannot be opened by other processes) with the process id of the process to be killed as part of its name. If the lock file cannot be created, another instance of `dumpproc` is dumping the process, and this instance of `dumpproc` aborts.

2. Kills the specified process with a `SIGDUMP` signal.

3. Reads `filesXXXXX`.

4. Resolves symbolic links for the current working directory and all open files.

5. If a file name points to a terminal, it is changed to `/dev/tty`, which will refer to the terminal of the process that will open it when the process is restarted.

6. For all local files, the string `/n/`*machine* is prepended to their names, where `/n/`*machine* is the name of the local machine. Local files are those whose name does not begin with `/n` after resolving symbolic links. This renaming ensures that the file names stored in `filesXXXXX` can be used to open the files of the migrated process as remote files from any machine on the network.

7. Rewrites `filesXXXXX` using the modified data.

`restart` uses the `rest_proc` system call to restart a process. It performs the following steps.

1. Verifies the existence of the three dump files and checks their magic numbers to verify that they have the appropriate format.

2. Reads the old user credentials from `stackXXXXX` and establishes them as its own.

3. Reads the old current working directory and establishes that as its own.

4. For each of the files contained in `filesXXXXX`, the program reads the information contained there. Files are opened with the correct access modes and their file pointer is positioned to the correct offset. Pipes, sockets, and files that no longer exist are redirected to `/dev/null` so that the restarted process finds an open file where it expects one. Standard input, output, and error output are treated specially; if the associated files cannot be be reopened, they are redirected to the terminal rather than the null device, so that the user can have some control over the restarted program. All files are reopened so that the same file descriptor is assigned to the same file as in the original process.

5. Reads the old terminal flags and sets those of the current terminal appropriately.

6. Calls `rest_proc` to restart the old program.

2.5. Implementation

A goal in this implementation was to minimize the number of changes to the kernel without sacrificing performance. For example, we could have implemented a more elaborate system by making more radical changes to the kernel, or we could have avoided modifying the kernel, doing everything at the user level and suffered poor performance. Our process migration system required only a few hundred lines of kernel code, most of which were additions.

2.5.1. Kernel Modifications

The kernel does not keep enough information about a process' current working directory and open files to enable us to deduce in a non-trivial way the names of these files. Instead, the kernel maintains an *inode* [14, 15], which specifies where a file is located physically on disk. To overcome this deficiency, the kernel structures were augmented to includes the names of these files.

One of the most important structures in the kernel is the *user* structure. This structure contains all the swappable information about each process. A fixed-size character string containing the full path name of the current working directory was added to this structure. When processing the `SIGDUMP` signal, the kernel emits that name to the dump file. This field is updated after each successful call to the `chdir` system call, which changes the current working directory. If the argument to `chdir` is an absolute path name, the name is copied to the user structure. If the argument is a relative path name, the name is combined with the value of the old current working directory name in the user structure, resolving references to the current working directory, ‘‘.’’, and the parent directory, ‘‘..’’. The result of this combination is then copied to the user structure. This field is initialized on the first call to `chdir` with an absolute path name; the updating procedure is skipped if the field is uninitialized. Such a `chdir` call is made early in the UNIX boot procedure, and new processes inherit this field from their parent, so this field is maintained correctly for all processes.

Information about open files is contained in a fixed-size array of pointers to *file* structures, one for each open file. A pointer to a dynamically allocated character string containing the absolute path name of the file to which the structure refers was added to the file structure. Keeping this field up to date permits the kernel to emit the names of open files when it processes a `SIGDUMP` signal. Dynamically allocated strings were used instead of fixed-length strings because file structures are not swapped, and because there are many processes running at any time, with, usually, more than one open file each. Fixed-size strings would have had to be large enough to accommodate long paths even though most paths are usually short and would have wasted large amounts of kernel memory.

The field containing the path name of the open file is initialized after a successful `open` [13] (open a file) or `creat` [13] (create a file and open it for output) system call by calling the kernel's memory allocator. The memory is released when the file is closed by the `close` [13] system call. To insure that the pointer to the name is either null or a valid pointer, the function that allocates new file structures initializes this pointer to null. After initializing the field, the kernel copies the name of the file, which is supplied to the kernel as an argument of the system call, into the newly allocated space. Before copying, if the file name is a relative path name, the name is combined with that of the current working directory, to create an absolute path name.

2.5.2. Kernel additions

Implementing the `SIGDUMP` signal is simply a matter of writing the appropriate data from the augmented kernel structures to files. The code is similar to that for processing the standard UNIX signal `SIGQUIT`, which causes a process to terminate, dumping the state of the process in a file named `core`. This state information is a subset of the information needed to be dumped when processing `SIGDUMP`.

In standard UNIX, new processes are created by first creating an identical copy of an old process with the `fork` system call, and then overlaying the copy with an image of a new program with the `execve` system call. The image is obtained from an executable file. `execve` cannot restart a process that was dumped with the `SIGDUMP` signal, because `execve` initializes the stack and clears the registers. `rest_proc` overcomes this deficiency of `execve`. `execve` was modified to check a global flag that, if set, indicates that it is called from within `rest_proc`. If the flag is set, instead of allocating a default stack, `execve` allocates a stack with a size given in another global variable, which is set to be the size of the stack the process that is being restarted had when it was stopped.

With these modifications to `execve`, `rest_proc` performs the following actions.

1. It opens the `stackXXXXX` file, checks access permissions and verifies that the file has the correct format by checking the file's magic number.

2. Reads the user credentials and the size of the stack from that file.

3. Sets the global flag indicating process migration, and sets the global variable that indicates the desired stack size.

4. Calls `execve` to execute the `a.outXXXXX` file, with the environment set to null. The environment of the old process is stored in its stack, so it will be automatically restored when the stack is read.

5. Resets the variable indicating process migration.

6. Sets the user credentials to those already read. The `a.outXXXXX` file was executed using the old credentials, so that only the owner of the process or the superuser can use `rest_proc`.

7. Reads the contents of the stack and registers.

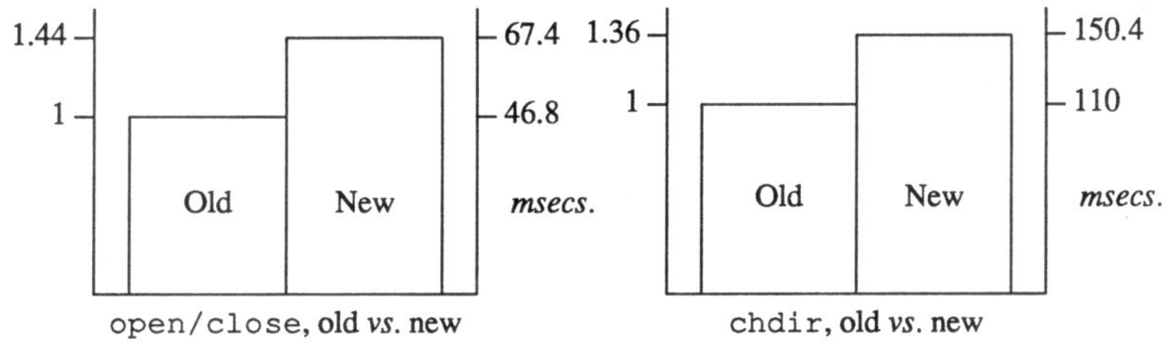
8. Reads the information on the disposition of signals and establishes it as that of the current process.

Using global flags to modify the behavior of `execve` is an error; doing so assumes that `execve` is atomic. But `execve` is not atomic; it is possible to interrupt `execve` while it is running on behalf of `rest_proc`, and then invoke it again in the normal way. In this case, the global flags are set incorrectly and `execve` will produce unpredictable results. This error can be corrected by rewriting `rest_proc` to include all of the code of `execve`, modified to initialize the stack and registers using the contents of the `stackXXXXX` file.

2.6. Performance evaluation

The modified `open`, `creat`, `close` and `chdir` system calls keep track of the names of the current working directory and all open files. For `open` and `close`, we measured this overhead by measuring the system CPU time of a program that opens and closes a file a hundred times. Measurements were taken both under the standard kernel and under the modified kernel. `creat` simply calls the same internal function that `open` calls, so it is unnecessary to measure the performance of `creat`. For `chdir`, we measured the overhead by measuring the CPU time of a program that executed one hundred sets of three calls to `chdir`, one with an absolute path name as an argument, one with the parent directory, ‘‘..’’, as an argument, and one with a path relative to the current directory, ‘‘.’’. These three sets were used in order to make certain that all cases of combining the new value of the current directory with the old one were considered.

These measurements are summarized in Figure 2.1; the performance of the standard kernel is normalized to 1 and is shown on the left vertical axis, and the actual times (average for one pair of `open/close` calls or a set of three `chdir` system calls) are shown on the right vertical axis. Our measurements show an overhead of about forty per cent (44% for `open/close`, 36% for `chdir`). The overhead is due to the string manipulation code that is executed by the modified system calls. The overhead for the `open` and `close` case is slightly more than for `chdir` because, in the former, the kernel also allocates and deallocates memory for the file name.



Performance of modified system calls

Figure 2.1

2.6.2. Dumping a process

Since the `SIGDUMP` signal is similar to `SIGQUIT`, it is appropriate to compare the performance of the former to that of the latter. To do this comparison, we started and killed a program repeatedly, in the following ways.

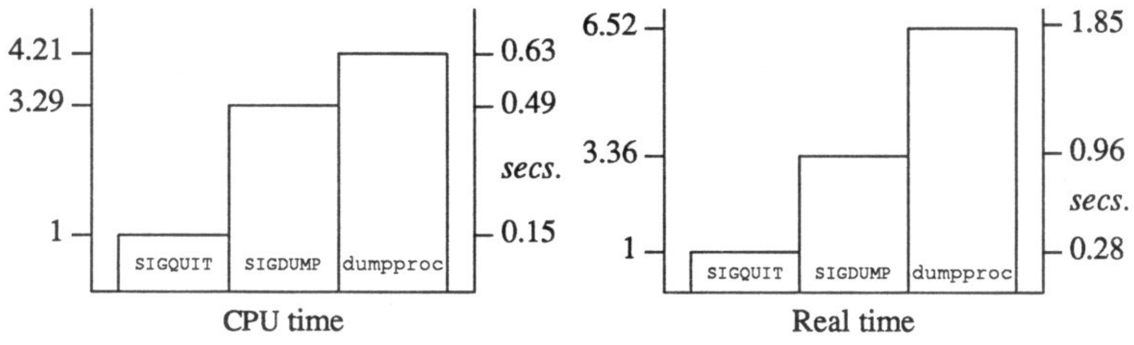
– By killing it with the `SIGQUIT` signal.

– By killing it with the `SIGDUMP` signal.

– By killing it with `dumpproc`.

We were measuring the performance of the process migration mechanism, not that of the file system, so we chose a program that was small, but that verified that our mechanism was working correctly. This program increments and prints three counters (a register, a static variable allocated on the data segment, and a variable allocated on the stack). On each iteration, the program inputs a line and appends it to an output file. The program was always killed after its first prompt for input.

In each case, we measured the CPU and real time required to kill the program. The results are summarized in Figure 2.2, where the performance of the `SIGQUIT` signal is normalized to 1. `SIGDUMP` requires roughly three times as much time (both CPU and real) as `SIGQUIT`. `SIGDUMP` must execute the code that dumps a file three times whereas `SIGQUIT` executes that code only once. `dumpproc` requires roughly four times as much CPU time and six times as much real time as `SIGQUIT`. The extra CPU time is expected, since `dumpproc` uses `SIGDUMP` to kill the process and modifies the `filesXXXXX` file. The large discrepancy between CPU and real time can be explained by noting that the three files produced by `SIGDUMP` are created by the process that is being dumped. When `dumpproc` tries to open the `a.outXXXXX` file, it must wait until the kernel context switches to the process being dumped so that the file can be created. After the file is created, `dumpproc` must wait again until the kernel context switches back to `dumpproc`. To avoid busy loops, `dumpproc` sleeps for one second after each unsuccessful open of `a.outXXXXX`. The performance of `SIGDUMP` is adequate; the order of magnitude of the CPU times for killing a process with `SIGDUMP` or with `dumpproc` is the same as for killing a process with a UNIX signal.

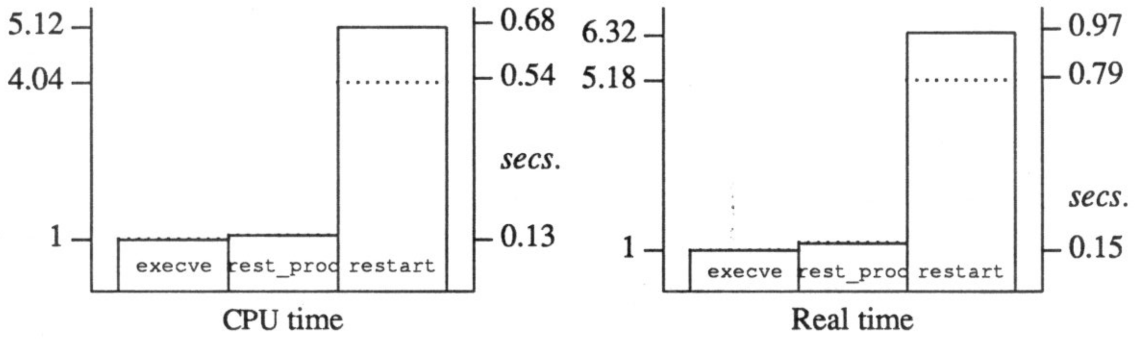


Relative performance of the `SIGQUIT` and `SIGDUMP` signals
and the `dumpproc` application

Figure 2.2

2.6.3. Restarting a process

Since `rest_proc` is similar to `execve`, it is appropriate to compare the performance of the two, along with `restart`. For each case, we measured the CPU and real time required by the system call or program in question to restart a test program (for `execve` we measured the time required to execute the `a.outXXXXX` file). The measurements were obtained by adding timing code to the kernel because these system calls destroy the process that invoked them. The performance of `restart` was measured by timing the execution of the command up to the point where it called `rest_proc`, and adding to this time the value already obtained by timing the system call. The results are summarized in Figure 2.3, where the performance of execve has been normalized to 1. The dotted line in restart's bar denotes the relative contributions of restart and rest_proc.



Relative performance of the execve and rest_proc system calls and the restart application

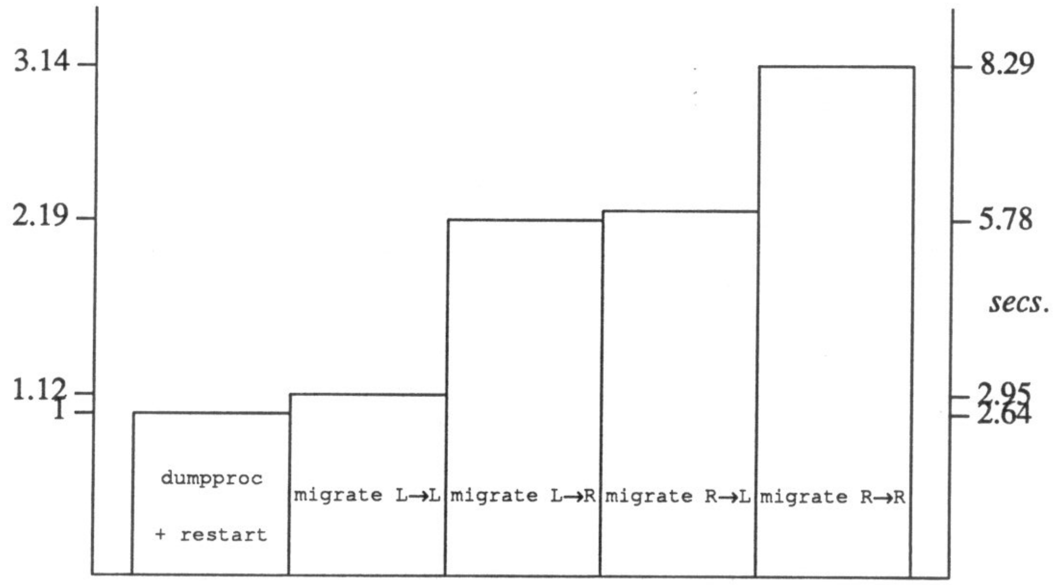
Figure 2.3

rest_proc takes only slightly longer than execve, as expected. restart takes significantly longer (roughly five times more CPU time and six times more real time) than execve. restart must check the existence and verify the format of the three dump files and, most importantly, initialize a part of the process environment at user level, including the open files, which requires a large number of open calls. Considering the amount of work that is being done, the delay is expected, especially since the unit of measurement is the time required to execute a process, which, for the test program was less than 0.2 seconds both in real and CPU time.

2.6.4. Migrating a process

migrate executes dumpproc and restart internally with the UNIX remote execution command on if any of those programs needs to be executed on a remote machine. on was used instead of rsh, the BSD remote shell command, because rsh can take three times longer than on to establish a connection between two machines. This additional delay would make migrate too expensive. on requires a certain amount of time to establish a connection with another machine, too, so, depending on where the process was originally running, and to where it is to be restarted, migrate can take as much as three times longer than it takes to run dumpproc and restart manually on the appropriate machines. In the worst case of migrating a process from a remote machine to another remote machine, which requires two remote executions, migrate takes a little over 8 seconds. In the more typical case of migrating a local process to a remote machine, migrate takes a little under 6 seconds. This delay is not insignificant, but is small enough to make migrate useful.

The measurement results are summarized in Figure 2.4, where the performance of the dumpproc/restart combination is normalized to 1. The similarity between manual migration and the local→local case and the nearly constant increase in response time whenever migrate executes an additional remote command suggest that much of migrate's overhead is due to on. The difference between the local→remote and remote→local cases is because, in each case, different programs are executed with a remote shell. In the first case, dumpproc is executed locally and restart remotely. In the second case, dumpproc is executed remotely and restart is executed locally.



Real time performance of the migrate application, compared to running the dumpproc and restart applications separately (L=Local machine, R=Remote machine)

Figure 2.4

2.7. Limitations

Not all processes can be migrated successfully under our current design. The main limitation is the inability to redirect pipes and sockets to a migrated process. For example, a process communicating with another via a socket will no longer be able to do so after it is migrated. The only thing we can do in our current implementation is to redirect socket I/O to a file, which is probably of little use. However, processes that are good candidates for process migration are those that have a large amount of CPU activity and little I/O activity. Such processes are often running by themselves without communicating with other programs.

Another limitation is that processes that wait for one or more of their children to complete cannot be migrated while waiting. When a process is moved to another machine, its children become orphans, and waiting for these children produces undefined results.

A partial solution to the problems above is to migrate entire groups of processes (e.g. the C compiler, which consists of a driver program, the various compiler passes, the assembler, and the linker). This migration could be done by first dumping the related processes to disk. Then, on the destination machine, a program would create as many children as the original parent process had, establishing necessary pipes for communication. Each of these child processes would restart one of the original child processes, and the parent would restart the original parent process. The following enhancements to our implementation need to be made in order to implement this scheme. File descriptors referring to pipes must be marked as such, along with an indication of which processes are communicating through that pipe. Processes that were in a wait state when dumped must be restarted in that state so that if a parent is waiting for its children to finish, it will continue to do so. Finally, the original process id of a child process must be restored, so that the id may be returned to the parent when the process completes its execution.

A more general way to deal with programs that use sockets is to use forwarding messages like those used in DEMOS/MP. On the original machine, the process is replaced with a process that receives messages from the sockets to which the original process was connected. The new process then forwards these messages to the migrated process. On the destination machine, the sockets of the migrated process are replaced with connections to the forwarding process on the original machine, thus reestablishing the connection between the migrated process and the processes with which it was originally connected. In addition to supporting these semantics, an implementation of this solution must prevent sockets from timing out. One disadvantage of this solution, however, is that if a process is migrated more than once, a number of forwarding processes will exist at the same time, thus decreasing the performance of the system. In addition to the overhead of these processes, interprocess communication for the migrated process will become slower, because messages must traverse a number of intermediaries.

Our system cannot handle processes with residual dependencies, which are processes that use knowledge of their environment, such as their process id, or the name of the machine in which they are running. An example is a process that repeatedly opens a temporary file whose name is built from the process id. After this process is migrated, its process id changes, and it will be unable to locate the appropriate file. This situation occurs if the program requests the process id from the system every time, instead of doing so only once and storing it in for subsequent uses. A more serious example is a process whose behavior depends on which machine it is running, e.g., consider a process that uses hardware floating point operations if running on machine A, or emulates them in software otherwise. If this process migrates from machine A to a machine that does not have hardware support for floating point operations, it may crash.

One solution to these kinds of problems is to add fields to the user structure for the old process id and the old host name, and change the `getpid` [13] (get process id) and `gethostname` [13] (get name of current machine) system calls to return these new values. This change would require new system calls that return the real values, regardless of whether or not the calling process has been migrated. Programs that use these new system calls know about the process migration mechanism and can avoid residual dependencies; Programs that use the old calls believe they are running on the original machine. This scheme eliminates the temporary file problem, but aggravates the problem of processes whose behavior depends on the machine on which they run. Using the floating point example, a process that starts on machine A will always believe that it is running on that machine, even if it is migrated. It will always use the hardware floating point operations, even if the machine on which it is running does not support them. However, there may be few applications that fall into this category, so this modification is worth considering.

Another solution is a system call that allows processes to notify the kernel that they can be migrated, and to migrate only processes that have issued this call. This approach works, but violates the requirement of transparency; only processes that are aware of the process migration mechanism can use it.

2.8. Process Migration Applications

Process checkpointing is a simple application to implement, requiring no additional code. Checkpointing is achieved by dumping the state of a process and then restarting it on the same machine. This capability can be used to take periodic snapshots of long-running programs. In case of a system crash, such programs can be restarted from the latest snapshot, rather than having to be run from the beginning.

An obvious application of process migration is load balancing where the average job's response time can be improved by moving processes from busy machines to idle ones. Good candidates for moving are jobs with a large expected running time, e.g., the CPU hogs mentioned in Reference 16. We have written two load-balancing applications, `move` and `dehog`.

`move` moves a process specified by the user to the least loaded machine in the system. It works as follows.

1. The least loaded machine in the network is located, which is done by comparing the current system load with that of all the other machines in the network. The system load is the average length of the run queue and is a statistic gathered by the kernel. `move` determines the current load by probing the memory of the kernel, and the remote load by reading the information that is broadcast to all machines in the network by the background process, `rwhod` [12], which runs on each machine.

2. If the current load is greater than the load of the least loaded machine by at least one, moving a job to that machine will reduce the load imbalance. If so, the process is moved to that machine after getting verification from the user. Verification is needed because the least loaded machine may be inappropriate for that process, or because moving processes to that machine may be disallowed.

`dehog` is similar to `move`, except that `dehog` presents a list of processes to move. This list is constructed by executing a `ps aux` command, which shows the percentage of CPU time that each process is consuming. The output is sorted on this number in decreasing value, so that ''hogs'' appear at the beginning of the list. For each process in the list, the user is queried about moving the process (certain programs, e.g., command interpreters, cannot be migrated). This query session continues until a process that is acceptable for migration is found. From that point on, `dehog` works like `move`.

`move` and `dehog` need manual intervention. A load imbalance must be detected by a human, who must then run these programs to rectify the imbalance. Even if the load is balanced after this intervention, another imbalance may soon occur as new jobs arrive. A superior solution is to automate the process one step further by having a process, running in the background, that detects a load imbalance and takes steps to reduce it. In the next Chapter, we describe such a load-balancing system and the outcome of a set of load balancing experiments.

CHAPTER 3

Process Migration versus Initial Placement

3.1. Introduction

Load-balancing strategies can be either non-preemptive or preemptive. Determining which of these two strategies yields the best load-balancing implementation is controversial; the literature provides arguments in favor of both.

Cabrera [17] argues in favor of process migration by studying system traces that show that most processes are short-lived. Therefore, an initial-placement algorithm can be of little benefit because these processes will incur the overhead of initial placement. On the other hand, a process-migration algorithm can perform better because the overhead will affect only the relatively few long-lived processes. It is therefore argued that one should implement a load-balancing scheme that identifies long-running processes and migrates them to other machines whenever this migration can improve the system-wide load.

Eager, Lazowska and Zahorjan [18] developed an analytical model that predicts the maximum benefits of initial placement and process migration by assuming that the overhead of load balancing is zero. They conclude that process migration will not yield significant performance improvements over initial placement. Only under extreme conditions can process migration offer performance improvements, and these benefits are expected to be modest. These extreme conditions are high values of the coefficient of variation (ratio of standard deviation to the mean) of the process service demand and the process interarrival time, or a workload without processes with intermediate service demands. According to this model, even if the overhead of process migration is considered, the results remain the same because initial placement is an inherently better strategy.

Leland and Ott [16] compared initial placement and process migration by conducting simulation experiments using system traces to generate a workload. These simulations suggested that either initial placement or process migration can produce significant improvements. Running the two algorithms together can produce an even greater improvement.

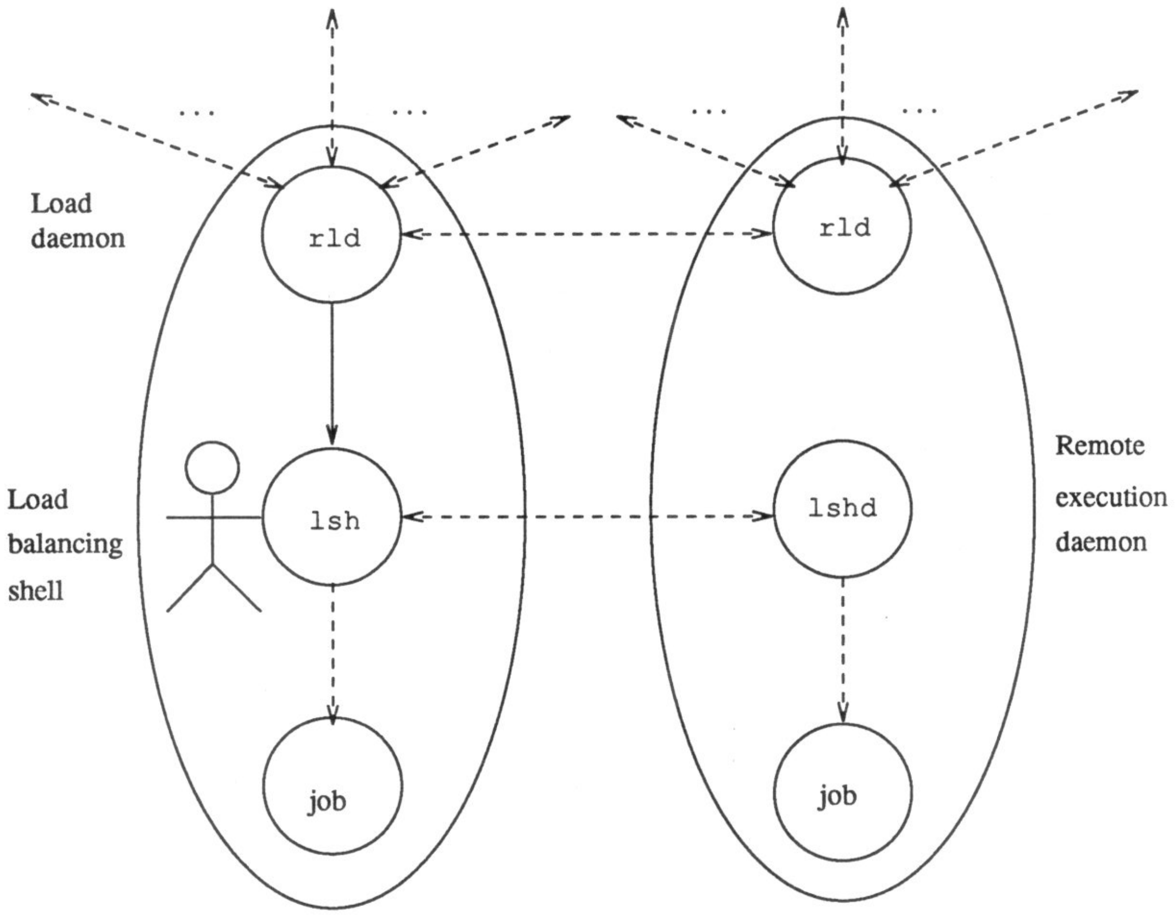
Finally, Krueger and Livny [19] conducted another set of simulation experiments to compare the two strategies. In these simulations, the workload was generated using a probability density function derived from the traces used in Reference 16. They concluded that using both process migration and initial placement can provide a significant performance improvement over using only initial placement.

It is not obvious which of the two load-balancing strategies, initial placement and process migration, is the better choice. Having implemented process migration, it was natural to try to answer this question by devising a set of experiments that use our system.

3.2. Implementation environment

The initial placement experiments were run using the load-balancing system described in Reference 20. This system provides a modified shell, `lsh`, that places new processes on the least-loaded machine in the network. On each machine, there are two background processes running assisting `lsh`. The first process, `rld`, is a remote load daemon that broadcasts and collects load information periodically. `lsh` determines the least-loaded machine in the network by consulting `rld`. If `lsh` determines that a process can be run locally, it simply runs it. If `lsh` determines that a process must be run remotely, it connects to a remote execution daemon, `lshd`, that is running on the remote machine. `lshd` starts the new process locally, establishing a connection between the process and the terminal on the machine where it was initiated. Figure 3.1 gives a schematic representation of this system. The dotted lines connected to `lsh` represent the two possible ways of starting a process with `lsh`: locally or remotely via `lshd`.

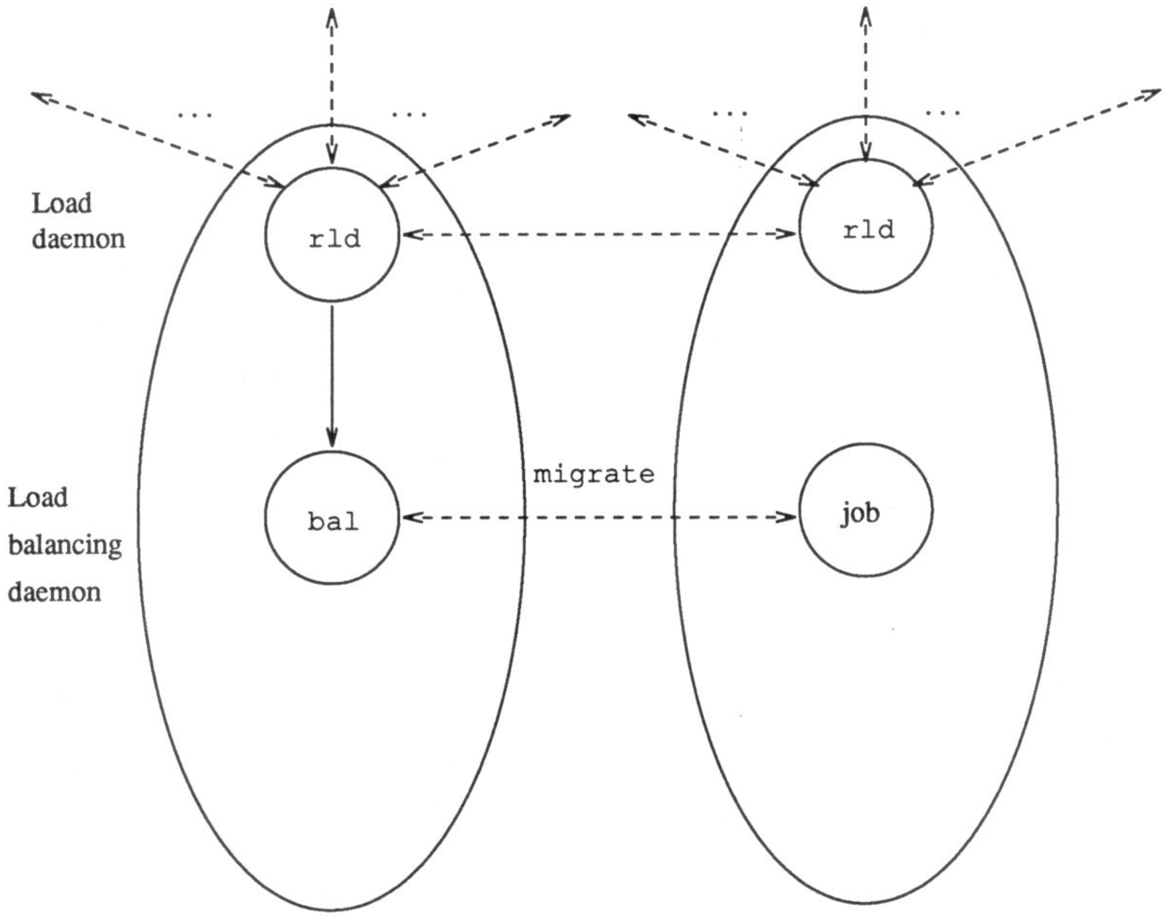
The load metric is the system load, which is the average number of processes in the run queue. For our experiment, processes were executed remotely only if the difference between the local load and that of the least-loaded machine was at least one.



Load balancing using initial placement.

Figure 3.1

The process migration experiments were run on a load-balancing system built using the process migration system described in the previous Chapter. This load-balancing system was built upon the initial placement system described above. There are again two background processes running on each machine. The first is `rld`, as described above, and the second is a load-balancing daemon, `bal`, that replaces `lsh` and `lshd`. `bal` checks the system load once a minute. If the load is greater than one, `bal` examines the list of local processes for one that has consumed more than a certain amount of CPU time. If one is found, `bal` moves it to the least-loaded machine using `migrate`. Figure 3.2 gives a schematic representation of this system. The process running on the right machine has been migrated by `bal` from the left machine.



Load balancing using process migration.

Figure 3.2

3.3. Workload

For our load balancing experiments, we needed a realistic workload. The ideal situation would have been to run the experiments on a working system, measuring the changes in response time from the use of the two load-balancing strategies. Unfortunately, using such a workload was not possible in our case. The available machines had a low actual workload because they had only few users. Moreover, limitations of the two load-balancing implementations forbid using an actual workload. In the initial-placement system, `lsh` is not interactive and must be invoked each time a process is to be started. This limitation makes it difficult to replace the standard shell with `lsh`. In the process-migration system, some processes cannot be migrated, as detailed in

Chapter 2. For these reasons, we used a synthetic workload.

Krueger and Livny [19], by applying weighted least squares analysis on the data collected by Leland and Ott [16], determined that the service demand of a typical UNIX process in CPU seconds has the following probability density function:

$$f(x) = .79(e^{-x/.31}/.31) + .192(e^{-x/2.8}/2.8) + .018(e^{-x/27}/27)$$

The mean service demand, $\overline{X}$, for this density function is 1.27 seconds, and the coefficient of variation, C_x, is 5.3. They generalized the above density function to form a family of functions sharing C_x, but varying in $\overline{X}$:

$$f(x) = (.79/.243\overline{X})e^{-x/.243\overline{X}} + (.192/.22\overline{X})e^{-x/.22\overline{X}} + (.018/21.2\overline{X})e^{-x/21.22\overline{X}}$$

Integrating from 0 to $+\infty$, yields the distribution function

$$F(x) = 1 - 0.79e^{-x/0.243\overline{X}} - 0.192e^{-x/2.2\overline{X}} - 0.018e^{-x/21.2\overline{X}}$$

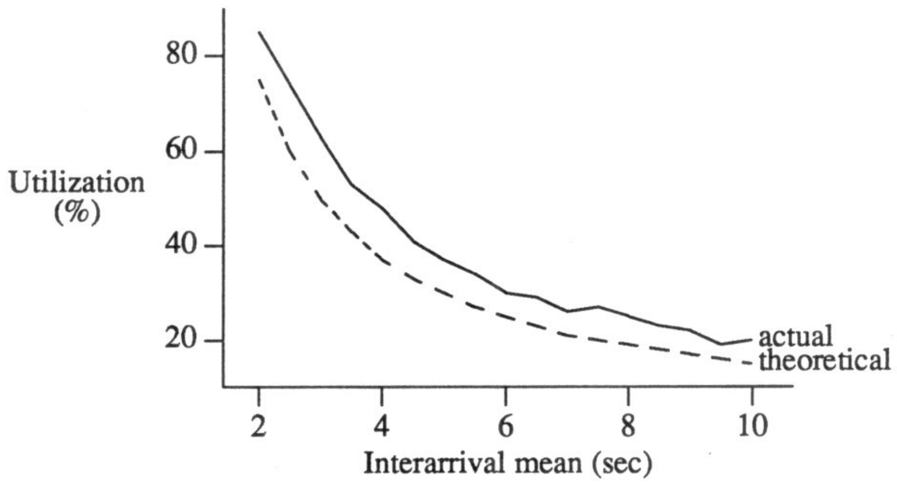
We used a synthetic workload generator to create random values that obey this distribution.

We can study the impact of the overhead of the load-balancing implementation by selecting different values of $\overline{X}$. This overhead is not a quantity that can be varied under real conditions, but this approach allowed us to treat it as such. Low values of $\overline{X}$ make our relatively high overhead comparable to the running time of the average process in the system, which makes the overhead a considerable factor in a load-balancing experiment. High values of $\overline{X}$ make the overhead small compared to the running time of the average process, which makes the overhead negligible.

To generate the workload, we must also generate the interarrival time between processes; we chose a simple exponential distribution

$$f(x) = 1/t_m \, e^{-x/t_m}, \qquad F(x) = 1 - e^{-x/t_m}$$

where t_m is is the mean interarrival time. Different values of t_m correspond to different utilizations for the machine. The utilization can be determined by either $\mathit{utilization} = \overline{X}/t_m$ or experimentally. We measured the utilization by running the workload using different values of $\overline{X}$ and t_m. For this measurement, we computed the CPU utilization by reading the value of `cp_time` from the 3.3 Sun O.S. kernel in a manner similar to that used by `vmstat` [12]. The experimental approach is more accurate because $\overline{X}/t_m$ does not include the system processes that are running on a machine concurrently with the workload. Because of these processes, the actual utilization is higher. Figure 3.3 illustrates the difference between the two utilization estimates.



Theoretical and actual CPU utilization
as a function of the interarrival mean, for $\overline{X} = 1.5$.

Figure 3.3

Finally, to complete our workload model, we need to model idle machines. If all machines are busy with a similar workload throughout an experiment, load balancing is unnecessary. For these experiments, we ran the synthetic workload on five machines, and reserved two machines as cycle servers. On each machine, the workload was generated using a different seed for the random-number generator, which yields some variance in the workload.

As an alternative, we could have used the distributions given by Mutka and Livny [21]. By studying traces measured on workstations, they derived distributions for the time a machine is available and unavailable. A machine was considered unavailable if it was being used or if it had been used recently. Using this approach, we could have generated the workload only when a machine is unavailable. However, since these distributions were derived on a MicroVAXII and the workload distributions were derived on a VAX 750 and a 780, it seemed inappropriate to mix the two.

3.4. The Experiments

We varied the importance of the load-balancing overhead by varying the mean service demand $\overline{X}$. We used three different values of $\overline{X}$: 1.5, 5 and 25 seconds. These are arbitrary values. The first is similar to the value of 1.27 seconds reported in Reference 19, the second is comparable to the measured overhead of our implementation†, and the third is an order of magnitude higher. For the mean interarrival time t_m, we chose three values that produced an average CPU utilization of 25, 50 and 75 per cent. Combining these two sets of values produced nine different experiments. Each experiment was run once without any load balancing to obtain the original workload, once with initial placement, and once with process migration, for a total of 27 experiments. Each of these 27 experiments was run three times to obtain a greater statistical significance. The workload for each of these experiments was generated by a background program that sleeps until it is time for a new process to arrive. The program then wakes up and starts a small program containing a timed loop that consumes the required amount of CPU time. For no load balancing and process migration, this small program was run locally. For initial placement, it was run at the appropriate machine using `lsh`.

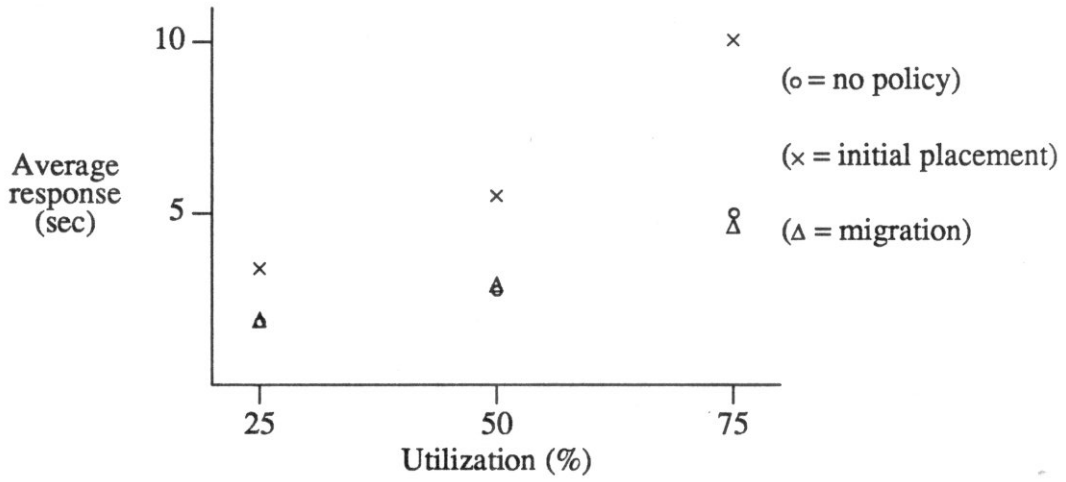
In the experiments involving process migration, we migrate processes that have consumed more than a certain amount of CPU time. For $\overline{X} = 1.5$, we migrate processes that consumed more than 16 seconds of CPU time. Sixteen seconds was reported initially by Cabrera [17], who found that only 2% of the processes in typical UNIX systems consume more than 16 seconds. This observation also held for our artificial workload with $\overline{X} = 1.5$, so we used this value to identify long-running processes, and only about 2% of all generated processes were process migration candidates. For $\overline{X} = 5$ and 25, this time was scaled proportionally to 53 and 266 seconds, respectively. For $\overline{X} = 1.5$ and 5, each experiment was run for 20 minutes, and for $\overline{X} = 25$, each experiment was run for one hour, in order to collect sufficient data. In the process migration experiments, only processes for the timed loop program were candidates for process migration so that

† The overhead of executing a process remotely in our initial-placement implementation is about 3 seconds [20]. The overhead of migrating a process to a remote machine is about 6 seconds (see Chapter 2).

programs that should not be migrated, like the workload generator, were not moved.

3.5. Results

Figure 3.4 shows the results for $\overline{X} = 1.5$.



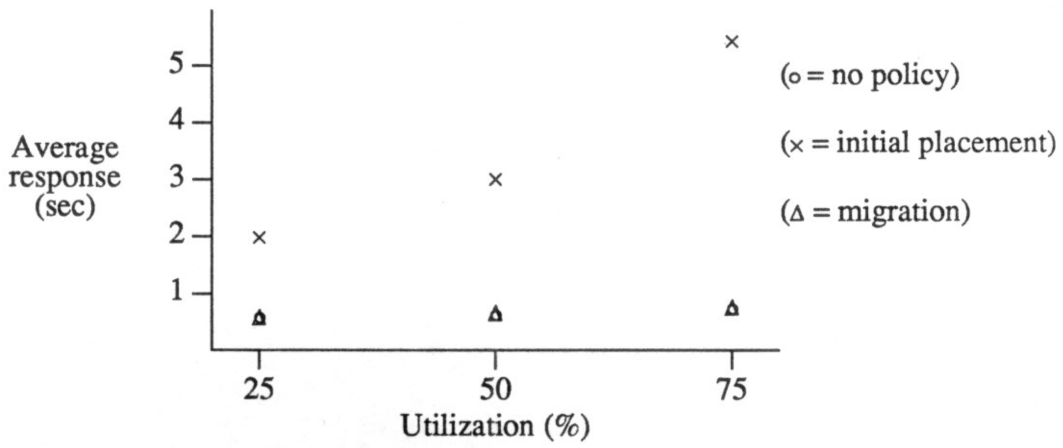
Average response for $\overline{X} = 1.5$

Figure 3.4

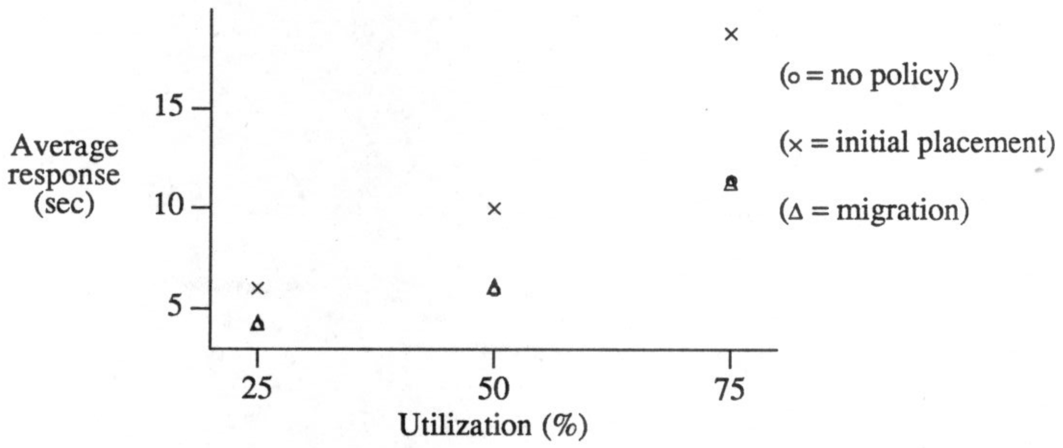
These experiments show that if the overhead is considerable, initial placement degrades the average process response time instead of improving it. Process migration produces results similar to the original workload, even making a marginal improvement for high CPU utilization. This behavior can be explained as follows. For initial placement, each arriving process incurs the initial placement overhead. Processes that are started locally incur less of this overhead because they are delayed only until the system determines that they can be executed locally. Processes that are executed remotely also pay the overhead of remote execution. The probability that a process will run remotely is high, so most processes pay the remote execution overhead. (Once load balance has been achieved, the probability of running a process remotely is $(n-1)/n$, where n is the number of machines.)

Process migration affects only few processes. Therefore, overhead plays a less significant role, and most processes run as if load balancing did not exist. When the CPU utilization is high, it is possible to improve response time slightly, despite the large overhead.

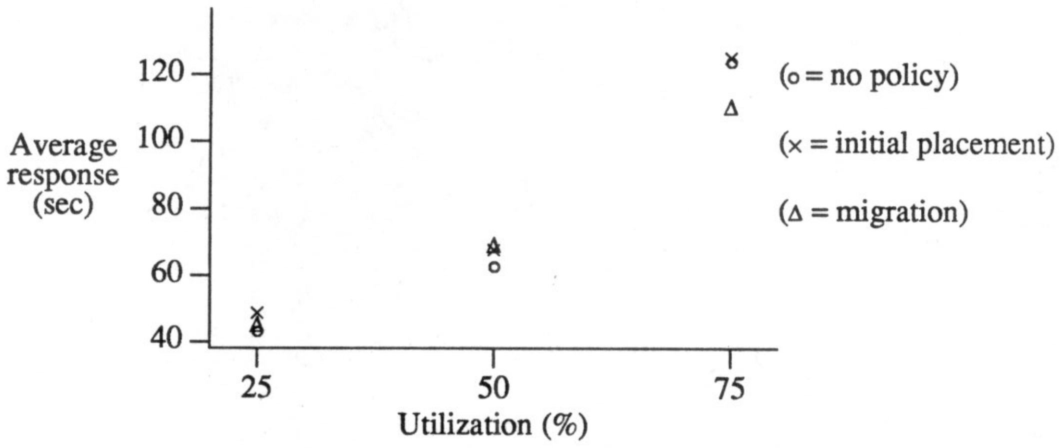
Figure 3.5 presents the same results showing the average response time of processes with small, medium and large service demands (small, medium and large processes). These three graphs were derived by averaging separately the response times for each process category. Small processes had a service demand of up to 1 second, medium processes had a service demand between 1 and 16 seconds, and large processes had a service demand greater than 16 seconds. In Figures 3.7 and 3.9 where similar graphs are shown for different values of $\overline{X}$, these intervals are scaled proportionally to $\overline{X}$.



Average small process response for $\overline{X} = 1.5$



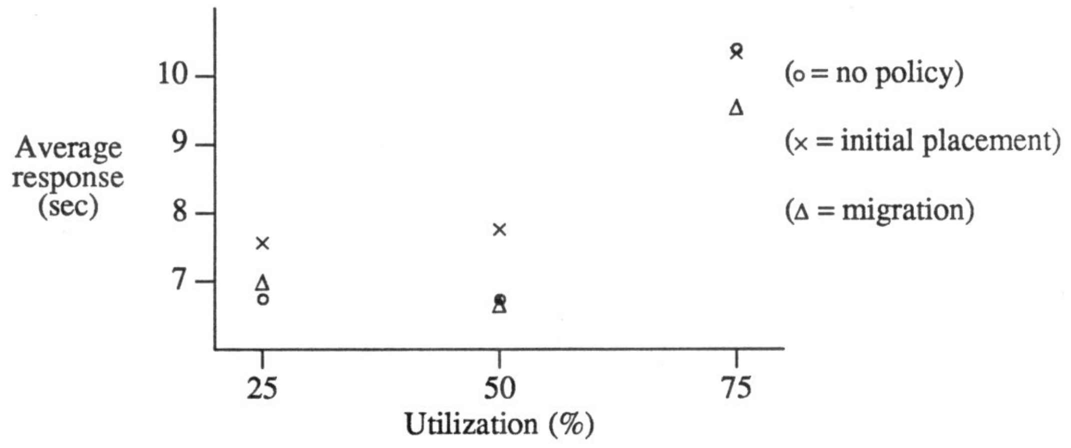
Average medium process response for $\overline{X} = 1.5$



Average large process response for $\overline{X} = 1.5$

Figure 3.5

Figure 3.6 shows the results for $\overline{X} = 5$.

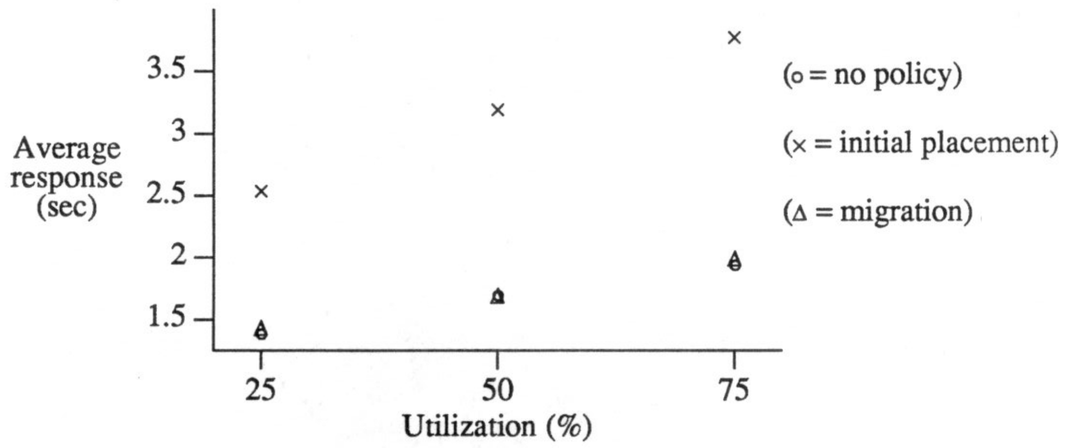


Average response for $\overline{X} = 5$

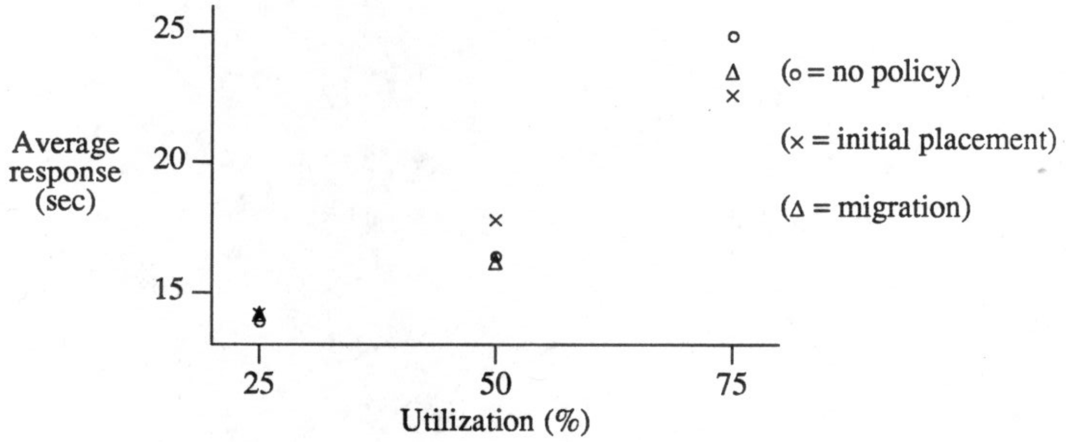
Figure 3.6

These experiments show that if the overhead is comparable to the mean service demand, for low and medium utilizations the results are similar to those for $\overline{X} = 5$. When the CPU utilization is high, initial placement is equivalent to no load balancing. Process migration, however, produces a noticeable improvement.

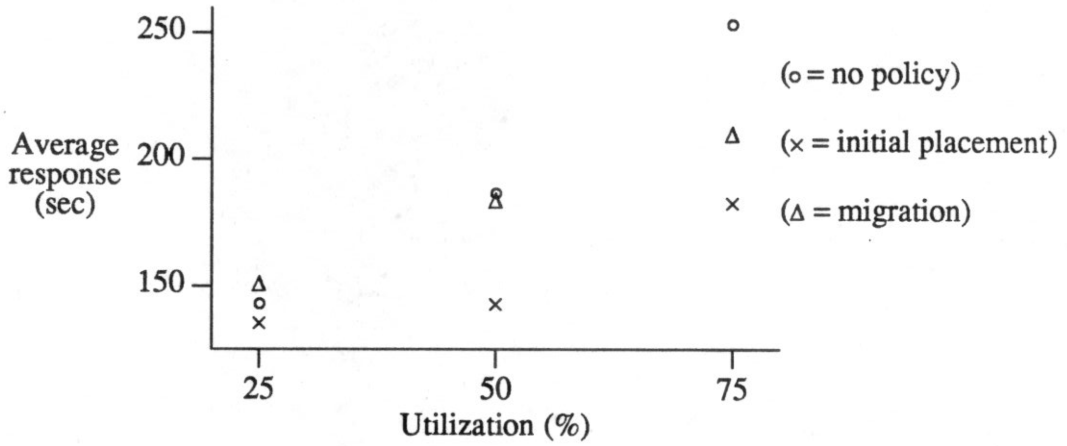
Figure 3.7 shows the same results for small, medium and large processes. Compared to the $\overline{X} = 1.5$ case, the situation appears different. Medium and large processes benefit from initial placement, because their service demand is comparable to or greater than the overhead. However, small jobs still suffer, so initial placement degrades the overall performance.



Average small process response for $\overline{X} = 5$



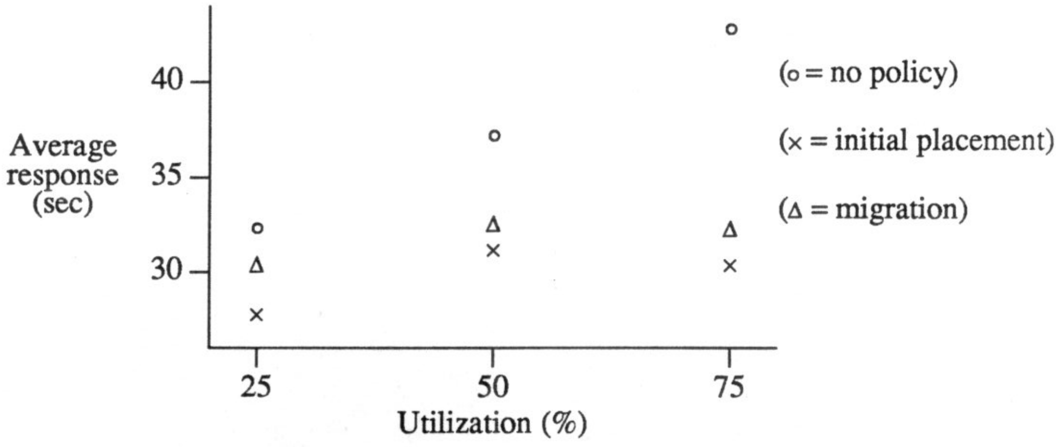
Average medium process response for $\overline{X} = 5$



Average large process response for $\overline{X} = 5$

Figure 3.7

Figure 3.8 shows the results for $\overline{X} = 25$.

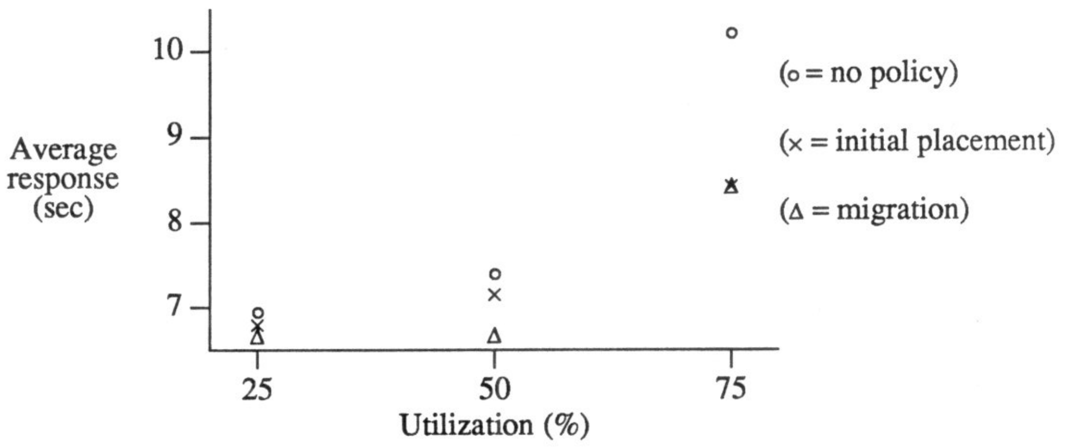


Average response for $\overline{X} = 25$

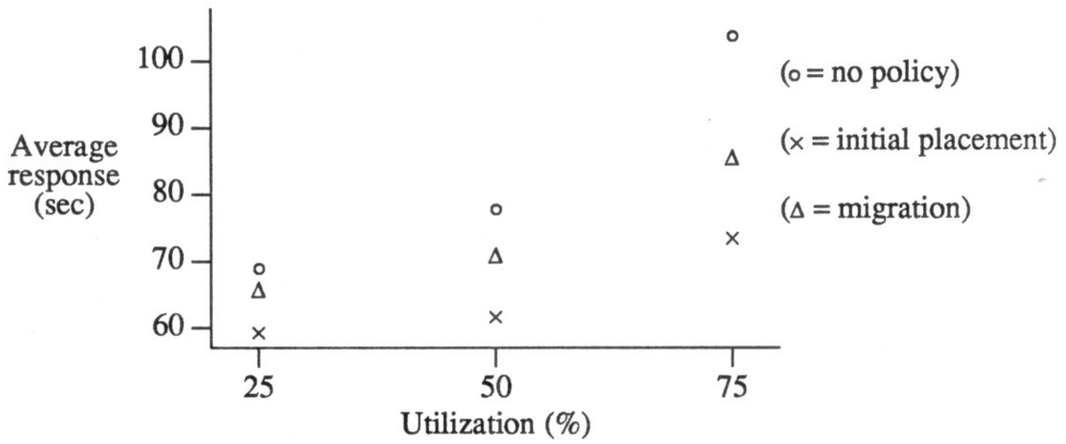
Figure 3.8

These results show that for insignificant overhead, the situation is reversed; both initial placement and process migration improve the average process response significantly. As suggested in Reference 18, there is little difference between initial placement and process migration, and initial placement appears to perform better. The overhead for load balancing is negligible in either case, so initial placement, which considers all processes, gives a greater improvement in response time than load balancing, which considers only a fraction of the processes.

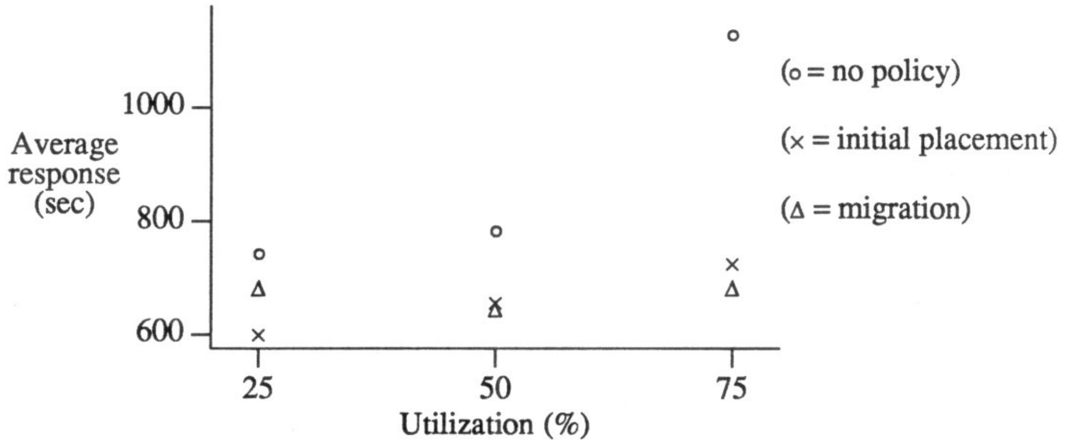
Figure 3.9 shows the same results, for small, medium and large processes.



Average small process response for $\overline{X} = 25$



Average medium process response for $\overline{X} = 25$



Average large process response for $\overline{X} = 25$

Figure 3.9

3.6. Conclusions

Our experiments comparing initial placement and process migration suggest that Eager, Lazowska and Zahorjan [18] are correct only in preferring initial placement when the overhead is insignificant. However, most implementations of initial placement have significant overhead, and our results show that process migration can be better in this case, especially if CPU utilization is high. If the overhead for either strategy is too high, load balancing is useless because the average response time of a process will either deteriorate or remain unaffected regardless of the strategy used. On the other hand, a good implementation of process migration can improve the average response time of processes running on heavily utilized machines. Our experiments suggest a policy that does nothing while CPU utilization is low to medium and uses process migration when the system load becomes high. Even if process migration runs continuously, regardless of the system load, no significant degradation will occur. Finally, even if the overhead for process migration is high, its use when CPU utilization is high may still yield small improvements in response time.

CHAPTER 4

A Dynamic File Caching Strategy

4.1. Introduction

The average response time of a process in a distributed system can also be improved with file caching. When a machine caches a file, the file is copied to disk, which avoids network delays associated with remote accesses.

Caching a file obviously speeds read accesses. However, writing to cached files introduces a new problem: if a machine updates only its own cached copy, other machines do not see those updates, thus resulting in an inconsistency. On the other hand, if a machine updates all copies of the file, write accesses become more expensive than they would have been if the file had not been cached because several remote updates are required instead of one. To take advantage of file caching and maintain file consistency, a caching strategy must balance the improved performance of read accesses against the degraded performance of write accesses in a way that the overall access performance is improved.

4.2. Existing file systems supporting caching

File caching is an old idea, but existing distributed file systems that support it deal poorly with maintaining file consistency. For example, in the Sun network filesystem, NFS [22], clients cache the buffers for a remote file in order to reduce the read and write requests to the file server. The cache is flushed when the file is closed. This scheme does not prevent inconsistencies. If a client updates a file, other machines do not see these updates while the file is open. Moreover, after the cache is flushed, other clients are not notified that the file has been changed on the server. These clients not only miss the update, but they will overwrite the server's updated copy with their own when they close the file.

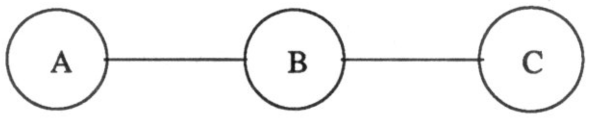
In the Andrew file system [23], reads and writes to a cached file are directed to the cached copy. As in NFS, the server's copy is updated only when the file is closed, resulting in similar inconsistencies. Unlike NFS, Andrew addresses the problem of updating the cache on other clients. A client's cache manager can request a *callback* from the server. The server will notify the client whenever a cached file has been updated so that the client can invalidate its cache. However, since the server's copy is updated only when a client closes the file, it is still possible to have inconsistencies between the server's file and the cached copy.

The Sprite network file system [24] maintains file consistency by not caching files that are opened for writing. Performance may suffer using this strategy, however, because the benefits of caching are lost.

4.3. Analytical modeling

The cost of accessing a file is not simply the cost of reading or writing a block from the machine accessing the file. We must also consider the cost of updating the cached copies on other machines and the cost that other machines pay to update the cached copy on that machine.

We define the cost of a single read or write to be equal to the distance between the machine where the file resides and the machine where it is accessed. The distance between two machines is the number of edges traversed in the network graph from one machine to the other. For example, in Figure 4.1,



A sample network with three machines

Figure 4.1

the distance from A to B is 1, and the distance from A to C is 2. In this example, the cost of doing a single read or write from A is zero if the file resides on A, 1 if the file resides on B, and 2 if the file resides on C.

The cost of a single access is f ; f_i refers to the value of f for the i^{th} machine, i.e., the cost of accessing a file on the i^{th} machine. If the i^{th} machine has more than one file, the cost of accessing the j^{th} file on the i^{th} machine is f_{ij}. Finally, $f_{ij-read}$ and $f_{ij-write}$

refer, respectively, to a read or a write access.

The cost of a read is obviously f. If the file is cached, the distance used to calculate f is 0. If the file is remote, this distance is the distance from the server. The cost of a write is $\sum_i^n f_i$, i.e., the sum of the cost of updating n copies of the file. This cost function is similar to that presented in Reference 25 and indicates that writes are more expensive when there is more than one copy of a file.

At each machine i, an open file j has a read/write ratio (r/w ratio) $\alpha_{ij} = r_{ij}/w_{ij}$, where r_{ij} and w_{ij} are the numbers of read and write accesses of file j from machine i in a given amount of time. The average cost of accessing file j from machine i is

$$c_{ij} = P(write) \times f_{ij-write} + P(read) \times f_{ij-read} \tag{4.1}$$

where $P(read)$ is the probability of doing a read and $P(write)$ is the probability of doing a write. Obviously

$$P(write) = \frac{w_{ij}}{r_{ij} + w_{ij}} = \frac{1}{\alpha_{ij} + 1} \tag{4.2}$$

and

$$P(read) = \frac{r_{ij}}{r_{ij} + w_{ij}} = \frac{\alpha_{ij}}{\alpha_{ij} + 1} \tag{4.3}$$

Using (4.2) and (4.3), (4.1) becomes

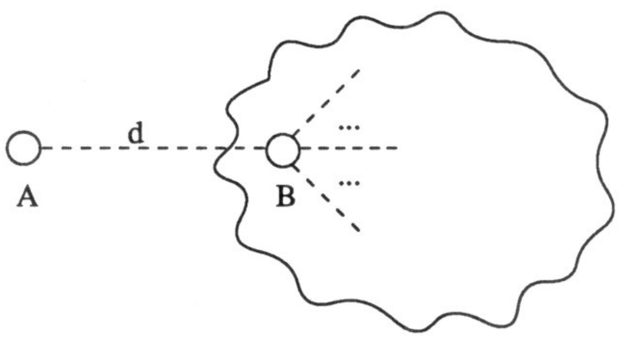
$$c_{ij} = \frac{1}{\alpha_{ij} + 1} \times f_{ij-write} + \frac{\alpha_{ij}}{\alpha_{ij} + 1} \times f_{ij-read} \tag{4.4}$$

Since a file can be accessed from many machines, a good caching strategy minimizes $\sum_i^n c_{ij}$, i.e., the sum of the average cost of accessing a file over all machines. One such strategy is as follows. Each time a machine must decide whether to cache a file or to destroy a cached copy, it examines this sum. If the file is not cached and the sum can be decreased by caching the file, the file is cached. If the file is cached and the sum can be decreased by flushing the cache, the cache is flushed. A good time to make these decisions is when the file is opened.

This strategy covers the general case, but is hard to implement because each machine must compute the average access cost for all machines in the network. Such a computation may be infeasible; it requires potentially a large amount of computation and that each machine knows the network topology and the r/w ratio of every file. Such data are unlikely to be available.

We can reduce this algorithm to one that makes caching decisions using only local information, independent of network topology, by assuming that the r/w ratio for a file is fixed throughout the network. This assumption is reasonable because most files belong to particular applications, and their r/w ratios are determined by those applications. For example, a travel agency database has a r/w ratio of about 1 [26]. Agents consult the database for the availability of tickets and then make reservations. A library database has a high r/w ratio because most requests are queries.

Consider the network in Figure 4.2.



Making a caching decision on machine A

Figure 4.2

We are about to decide whether to cache a file on A, using the algorithm described above. We have assumed that the r/w ratio of that file is fixed throughout the network and is equal to α. A will access the file from B, which is a distance d away. Assume that B is the closest machine to A with a copy of the file, and that updating a file that exists both on A and B is done by updating the file on B and propagating the update to A. There are n machines that have a cached copy of the file, and there are m machines that are accessing the file without having a cached copy.

If A does not cache the file, the cost of a read from A will be d, and the cost of a write from A will be $d + C_w$; C_w is the additional cost of updating the cached copies elsewhere. If A caches the file, the cost of a read from A will be 0, and the cost of a write from A will remain $d + C_w$.

At first glance, this reasoning seems to indicate that A should always cache the file because reads become cheaper, while the cost of writes remains unchanged. However, if the file is cached, the cost of writes will increase for the other machines that are using the file because these machines will each have another copy to update. Therefore, we need to consider the cost of an access from each of those machines as well as the cost from A.

First, we compute the cost of an access from each of the n machines that are caching the file. If A does not cache the file, the cost of a read from the i^{th} of those machines will be 0, and the cost of a write will be C_{iw}; C_{iw} is the cost of updating all other cached copies from the i^{th} machine. If A caches the file, the cost of a read from a machine with a cached copy will remain 0, and the cost of a write will become $C_{iw} + d$; d is the additional cost of propagating the update from B to A.

We must also compute the cost of an access from each of the m machines that are accessing the file without having it cached. If A does not cache the file, the cost of a read from the i^{th} of those machines will be C_{ir}, and the cost of a write will be C_{iw}. If A caches the file, the cost of a read from the i^{th} machine will remain C_{ir}, and the cost of a write will become $C_{iw} + d$. These computations assume the worst case ignoring the possibility that the i^{th} machine could read the file from A.

If the sum of the average cost of an access over all the machines that are accessing the file is less when A caches the file than when it does not, A should cache the file. Specifically, A should cache the file when

$$\frac{d + C_w}{\alpha+1} + \sum_i^n \frac{C_{iw} + d}{\alpha+1} + \left[\sum_i^m \frac{C_{iw} + d}{\alpha+1} + \sum_i^m \frac{\alpha C_{ir}}{\alpha+1}\right] \tag{4.5}$$

is less than

$$\left[\frac{d + C_w}{\alpha+1} + \frac{\alpha d}{\alpha+1}\right] + \sum_i^n \frac{C_{iw}}{\alpha+1} + \left[\sum_i^m \frac{C_{iw}}{\alpha+1} + \sum_i^m \frac{\alpha C_{ir}}{\alpha+1}\right]$$

or,

$$\sum_i^n \frac{d}{\alpha+1} + \sum_i^m \frac{d}{\alpha+1} < \frac{\alpha d}{\alpha+1} \tag{4.6}$$

or,

$$\frac{n\,d}{\alpha+1} + \frac{m\,d}{\alpha+1} < \frac{\alpha d}{\alpha+1} \tag{4.7}$$

or,

$$\boxed{\alpha > m + n} \tag{4.8}$$

A should cache a file if its r/w ratio is greater than the number of machines that are already accessing it. This result is independent of the network topology and requires only local information to make caching decisions. The r/w ratio for a file is often obvious from the application, or it can be determined locally from the file's past history of use. R/w ratios are assumed to be fixed, so the local r/w ratio is the r/w ratio for all machines.

4.4. Simulation experiments

We ran a set of simulation experiments to evaluate the algorithm described above, and to compare it with the simple alternatives of always caching and never caching. For the first of these two alternatives, once a file is cached, the cache is never flushed; for the second, remote files are always accessed from the server.

The simulations were written using the SIMPAS [27] simulation package. This package is a preprocessor for Pascal that provides extensions for writing simulations. These extensions are event scheduling, queues, and various random number generators. We simulated point-to-point and Ethernet-type networks. In the point-to-point case, we examined networks configured as rings, binary trees and fully connected networks. In the Ethernet case, we examined networks with and without broadcasting.

4.4.1. Point-to-point networks

For the point-to-point network simulations, we represented a machine as a disk queue and a network queue. Disk requests are serviced one at a time; each request consumes 20ms (a typical disk access time) after which the next request is processed. Messages are sent to other machines by inserting them in the network queue of the destination machine. This insertion occurs after a delay equal to the time required for a message to reach that machine. We used two different values for this delay, 20ms and 40ms.

Typical network hardware may be faster, but these delays account for software overhead. For example, the maximum transfer rate observed for `ftp`† transfers was about 50K bytes per second. The 20ms delay was derived from this rate by assuming that files have a block size of 1K and that one block is transferred per message. This transfer rate was the maximum observed, so we also used an ''average'' delay of 40ms. Finally, each machine consumes 1ms of processing time for each message in its network queue.

During initialization, each machine computes the minimum spanning tree of the network with itself as the root. Using these trees, each machine can determine the shortest path for a message. Processes arrive at each machine at an exponential interarrival rate with a mean of 2 seconds. This value was used in the load-balancing experiments for the case with a service demand of 1.5 seconds, which was similar to the case described in Reference 19. Each process performs 1 to 10 accesses on one of the files in the system; the number of accesses and the file is chosen randomly. Each access is a read or a write chosen randomly with a probability given by the r/w ratio α of the file being accessed. Since α is the same for all machines, $P(read) = \frac{\alpha}{\alpha+1}$. Each read or write accesses a random block in the 100-block file.

Before the first access, a process sends an open request to the server. The server responds with the number of machines using the file and a list of the machines that are caching the file. Based on the number of users, the process decides whether to cache the file (if it hasn't done so already), or whether to flush its cache (if the file has been cached). If the process creates or flushes the cache, the server is notified; the server updates its list of cache users and informs the users of the file of the change. When the server receives acknowledgement from these users, it, in turn, returns an acknowledgement to the original process.

Caches are created on demand, i.e., they contain only blocks that have been accessed at least once. Reads are directed to the cache if it exists and contains the requested block; otherwise they are sent to the server. Reads can be intercepted and

† `ftp` is the user interface to the ARPANET standard File Transfer Protocol and transfers files to and from a remote site [12].

serviced by a machine that has the requested block in its cache. Machines also populate their cache by copying blocks that are being transmitted through them, en route from the server to a reader.

Writes are made by requesting a block lock from all machines that have a copy of the file, writing the block, and releasing the lock.

4.4.2. Ethernet

The Ethernet simulations are a variation of the point-to-point network simulations. Instead of sending messages to the network queue of the destination machine, messages are sent to a central queue. One message in this queue is sent to the network queue of the appropriate machine every time period, which is equal to the network delay. This approach simulates the Ethernet property that all messages pass through the same hardware medium regardless of source and destination. For these experiments, we used network delays of 5 and 10ms. For larger values, the simulations overwhelm the system, producing average job response times greater than the mean interarrival rate. This problem occurs because all traffic goes through the central queue, which becomes a bottleneck. Small network delays permit the queue to accommodate the inflow of messages.

We studied variations of Ethernet with and without broadcasting. With broadcasting, a message is transmitted to multiple destinations by inserting one message in the central queue. When the message is processed, it is copied to the network queue of each destination machine. Messages can be transmitted to multiple destinations, e.g., when a block is updated on more than one machine.

In all cases, we compared the caching strategy suggested by inequality (4.8) to always caching and never caching. Below, we refer to the first of these strategies as ''our strategy''. These three strategies differ in the way file caches are created and destroyed.

Using our strategy, a machine compares the number of machines that are accessing the file, n, to the r/w ratio α of that file whenever a remote file is opened. If $n > \alpha$ and the file is already cached, the cache is flushed; if $n < \alpha$ and the file is not cached, a cache for that file is created. Note that n is the number of machines that are accessing the file when the file is opened and varies with time.

Always caching was implemented by modifying our strategy. When a machine opens a file that it isn't caching, it creates a cache for that file and never flushes it. Never caching was implemented by modifying our strategy so that machines never cache files.

4.5. Simulation results

All experiments were run for one hour of simulation time, which allows the average process response time to stabilize. Running the simulation longer did not change the average response time significantly. Each experiment was run three times using a different seed for the random-number generator. In the following Figures, never caching, always caching, and our caching strategy are denoted, respectively, by ''never'', ''always'' and ''cache''.

4.5.1. Ring configuration

Our first network has 5 machines configured as a ring, i.e., the i^{th} machine is connected to the $(i+1)^{th}$ machine modulo 5. Note that we are not simulating a token ring, but simply a network whose nodes are connected in a ring. We used network delays of 20ms and 40ms, placed all files on one machine, and varied the number of files from 1 to 5. The results are presented in Figures 4.3 and 4.4. In all cases, always caching produces worse results than never caching at low r/w ratios. Writes are so frequent that the overhead of updating multiple copies is more than the benefit of reading files locally. Always caching catches up with never caching at $\alpha=2$ for this network, and as α increases, always caching improves performance significantly over never caching. However, our caching strategy performs better than always caching. At low r/w ratios, our strategy performs as well, or slightly better, than never caching (except for a single file on a 20ms network). At middle r/w ratios, our strategy performs better than both always and never caching, and at high r/w ratios, our strategy is equivalent to always caching. As the number of files increases, the differences in the three algorithms become more prominent, especially at low r/w ratios. To understand this behavior, consider the case of one file and $\alpha = 1$. Using our algorithm, only one machine will have a cached copy of the file. As more files are introduced, additional cache copies are created (one for each additional file) and more machines benefit from caching.

4.5.2. Fully connected network

Our next network is a fully connected network of 5 nodes with all files on one machine. Again, we used network delays of 20ms and 40ms, and varied the number of files from 1 to 5. The results are given in Figures 4.5 and 4.6. In all cases, at low r/w ratios, the performance of all three caching strategies is similar. As the r/w ratio increases, our strategy and always caching offer significant improvements over never caching. At higher r/w ratios, and as the number of files increases, our caching strategy improves from being worse than always caching to becoming similar to it. Always caching is expected to be the best strategy in a fully connected network. Updating a file on more than one machine does not cost more than updating the file on just one because there is a direct path to all machines.

4.5.3. Binary tree

Our next network has 7 nodes configured as a binary tree of depth 2 with all files placed on the root. Again, we used network delays of 20ms and 40ms, and varied the number of files from 1 to 5. The results are presented in Figures 4.7 and 4.8 and are similar to those for the ring.

4.5.4. Ethernet without broadcasting

The next case is a network of 5 machines connected through a non-broadcasting Ethernet with all files on one machine. We used network delays of 5ms and 10ms, and varied the number of files from 1 to 5. The results are shown in Figures 4.9 and 4.10. For a 5ms delay, our caching strategy performs worse than never caching at low r/w ratios, except when there is only one file, in which case our strategy performs better than never caching. As the r/w ratio increases, our strategy catches up and surpasses never caching. Always caching has a similar behavior, but at low r/w ratios, it yields average response times greater than the mean process interarrival rate. For a 10ms delay, the network is too slow, and both our caching strategy and always caching produce much worse results than never caching, although our strategy does not overwhelm the system.

4.5.5. Ethernet with broadcasting

Our final network has 5 machines connected through an Ethernet with broadcasting. We used a network delay of 5ms. The results are shown in Figure 4.11 and are similar to the previous case, except that always caching does not overwhelm the system at low r/w ratios.

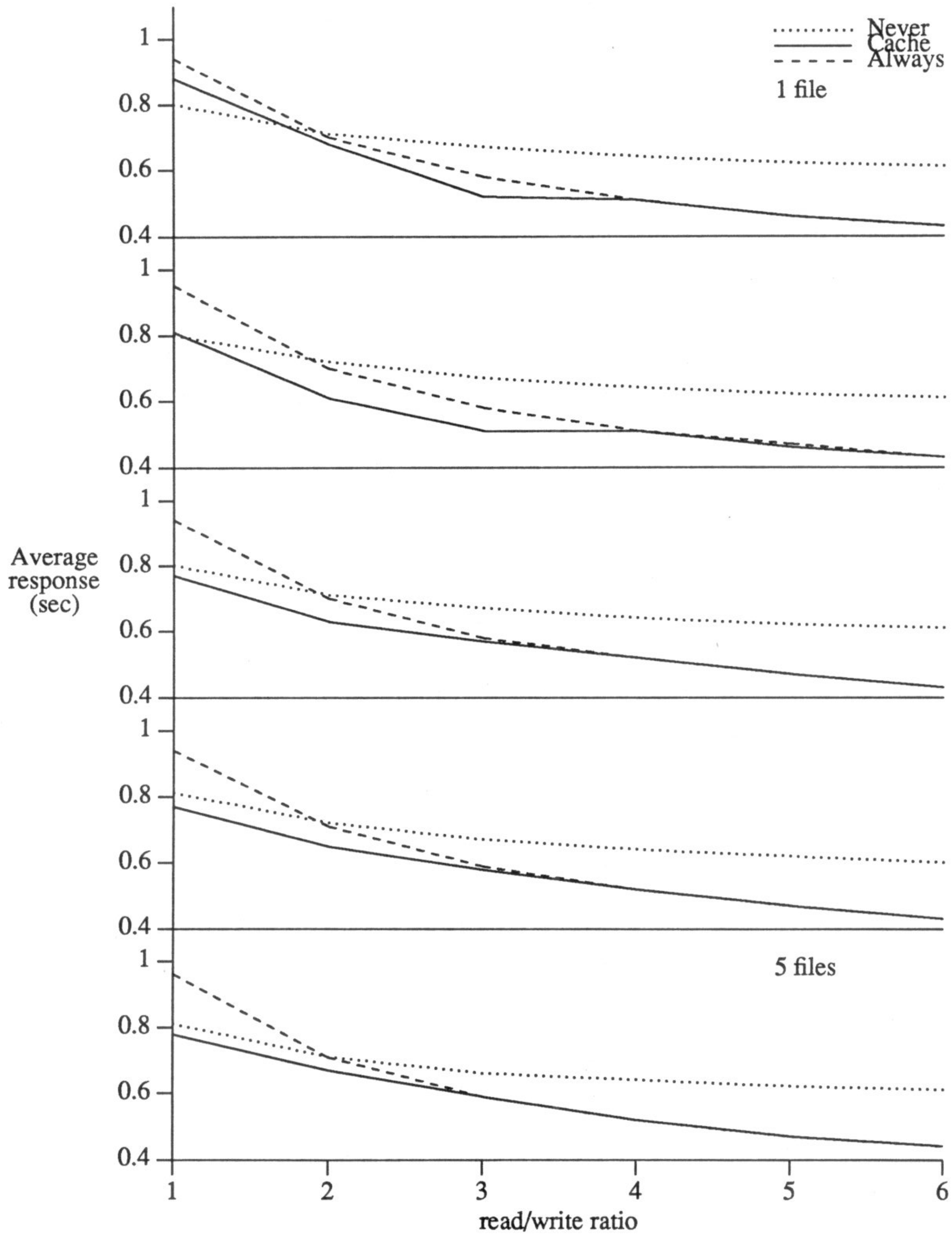
4.5.6. Larger networks

The networks examined above are small, so next we doubled the size of each network. We used a network delay of 20ms for point-to-point networks and 5ms for Ethernet networks. We used only these values because both 20ms and 40ms delays produced similar results in the point-to-point case, and always caching overwhelmed the system for a 10ms delay in the Ethernet case. For Ethernet, we examined only the case with broadcasting, because it is the most realistic. Finally, we only examined the case for 5 files because this case highlights the differences between the various caching strategies.

The results for the point-to-point networks are given in Figures 4.12, 4.13, and 4.14. These figures show the results for a 10-node ring, a 10-node fully connected network, and a binary tree of depth 3 (15 nodes), respectively. The ring and fully connected networks produced similar results as before. For the binary tree, however, never caching produced average response times larger than the mean process interarrival rate, overwhelming the system. At low r/w ratios, our caching strategy performs worse than always caching (even overwhelming the system for very low r/w ratios). At higher r/w ratios, our strategy is equivalent to always caching. It is clear that a binary tree with the file server at the root is a poor network design where caching is concerned.

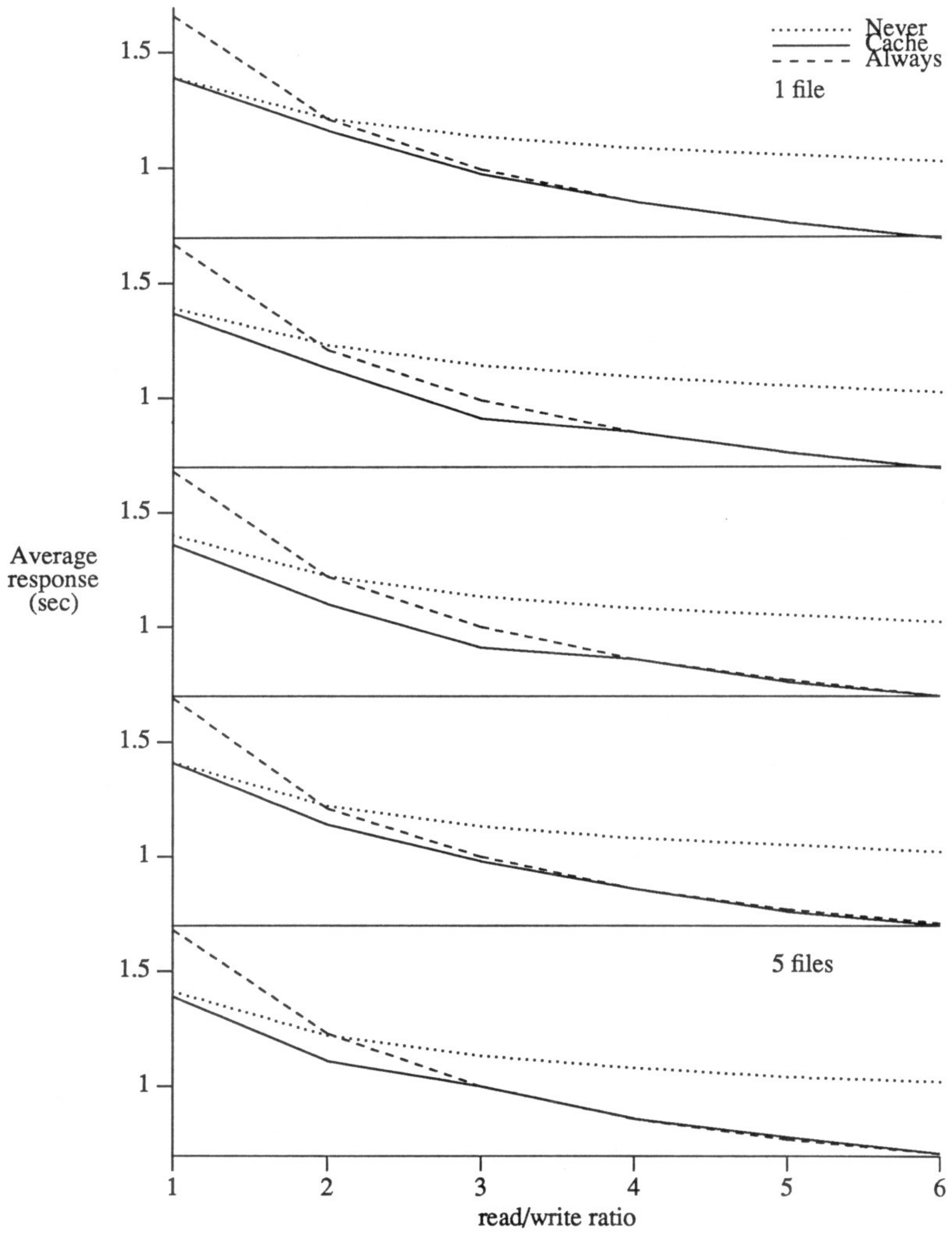
The results for Ethernet with broadcasting are given in Figure 4.15. These results are similar to those observed in the 5-node network. Both our caching strategy and always caching are inferior to never caching at low to medium r/w ratios, while, at medium to high r/w ratios, they are both superior to never caching.

Note the shape of the curve for our caching strategy in Figure 4.15. As the r/w ratio increases, so does the average job response time until a r/w ratio of 4. At this point, response time decreases and becomes less than the response time for never caching at a r/w ratio of about 6. At low r/w ratios, using our strategy yields only a small amount of caching. As the r/w ratio increases, so does the amount of caching. Finally, the hump in the curve occurs approximately at the same r/w ratio where always caching ceases to overwhelm the system. These observations suggest that for this particular network, and at low r/w ratios, any kind of caching increases process response time.



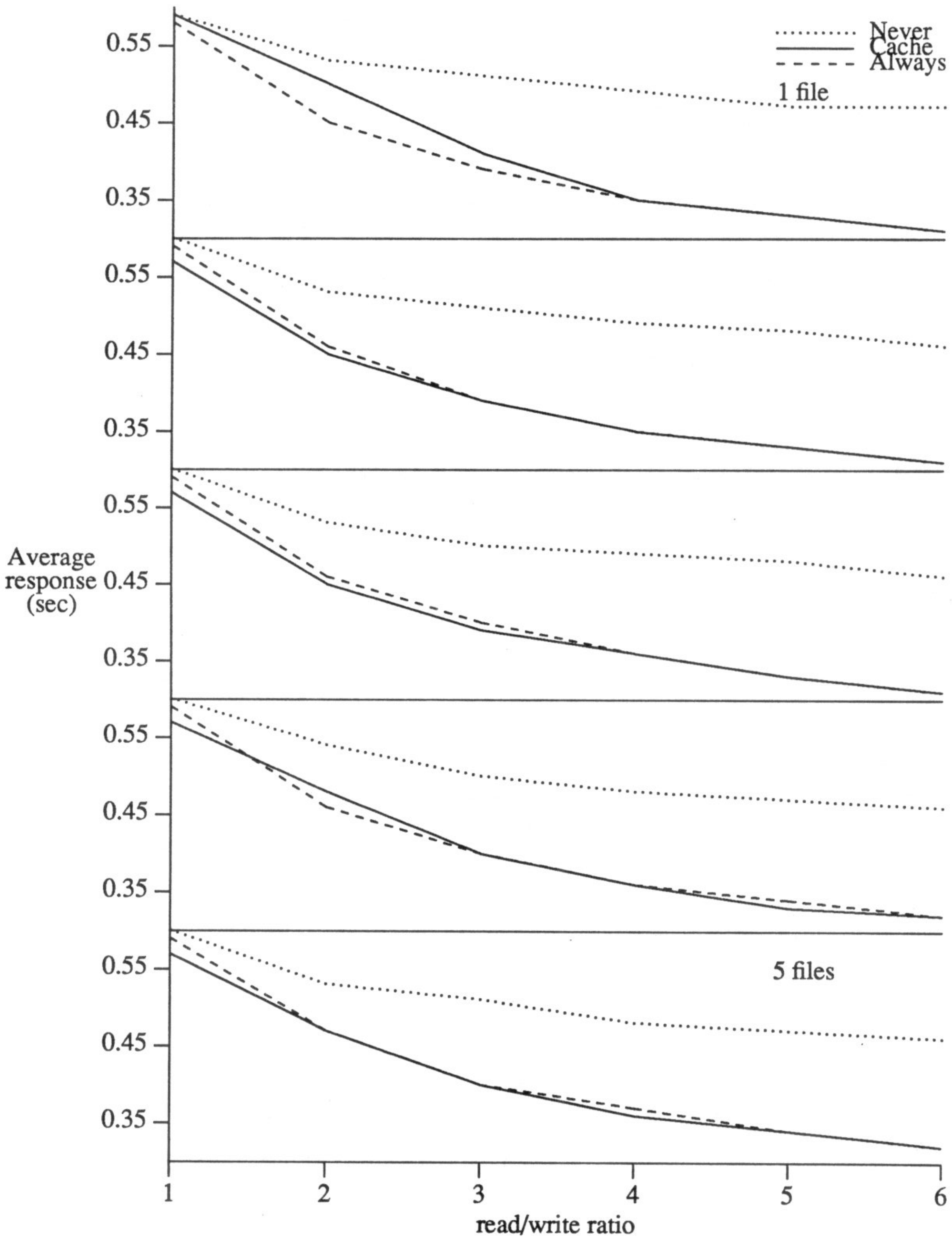
Ring configuration with 5 nodes, 1-5 files, 20ms network delay

Figure 4.3



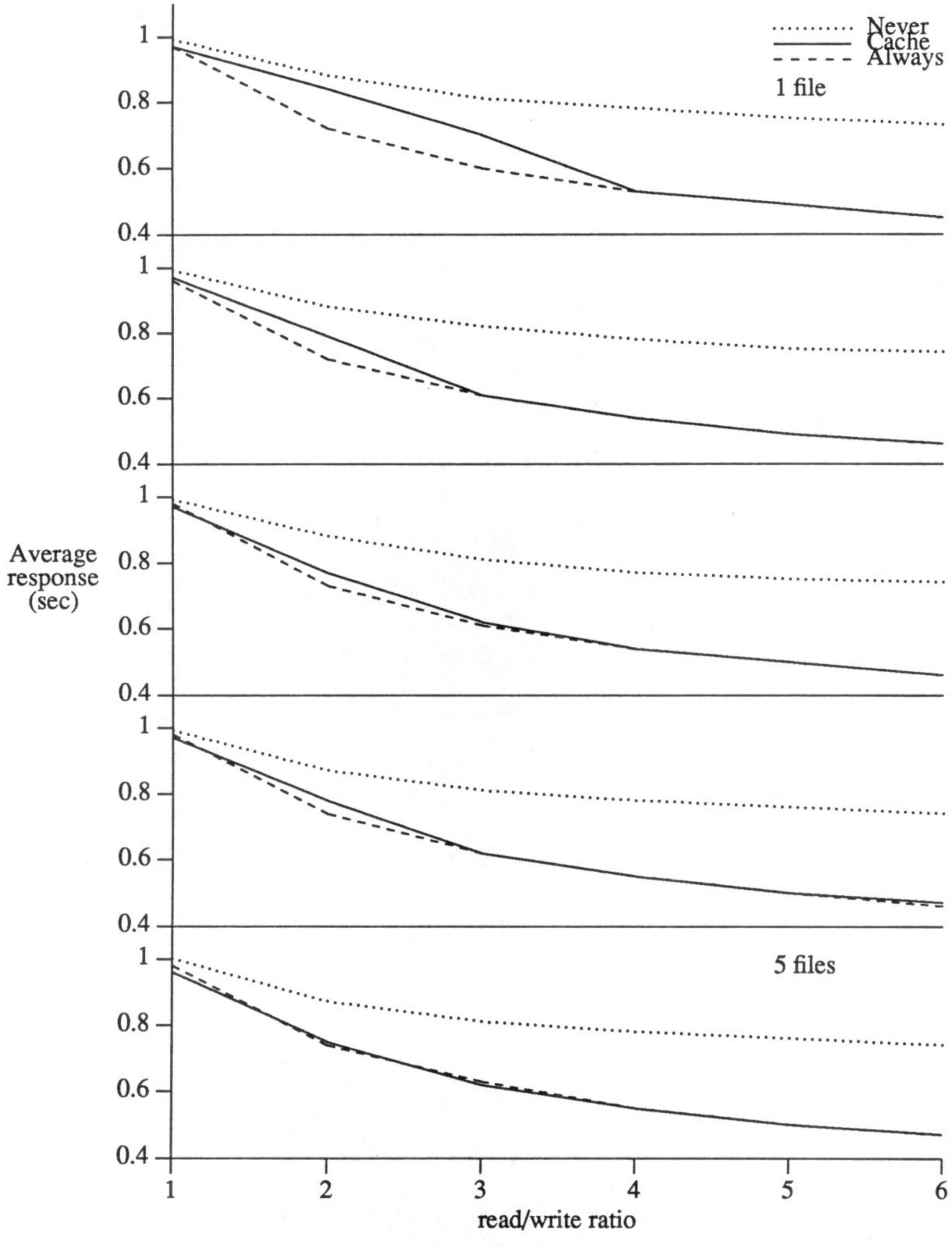
Ring configuration with 5 nodes, 1-5 files, 40ms network delay

Figure 4.4



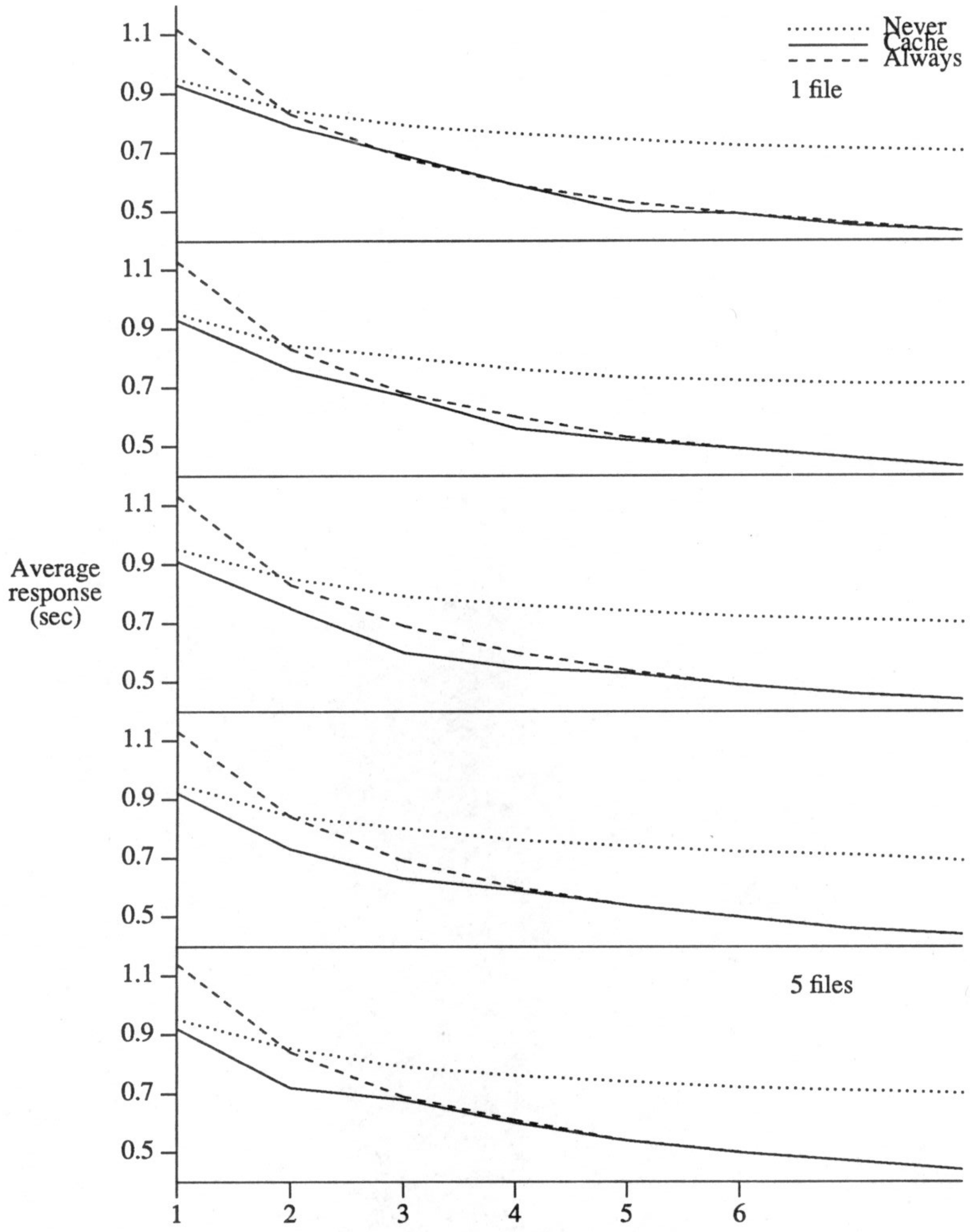
Fully connected network with 5 nodes, 1-5 files, 20ms network delay

Figure 4.5



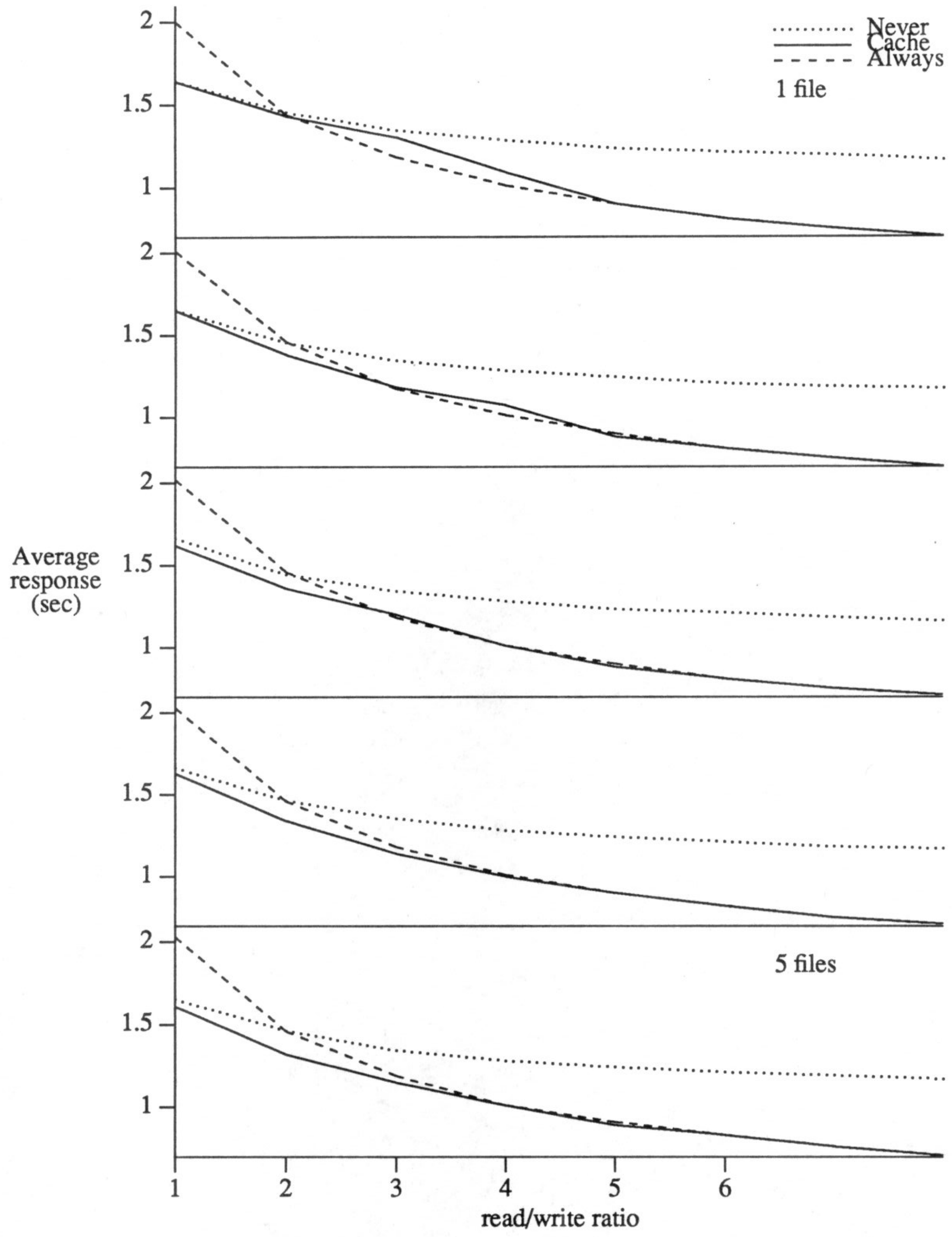
Fully connected network with 5 nodes, 1-5 files, 40ms network delay

Figure 4.6



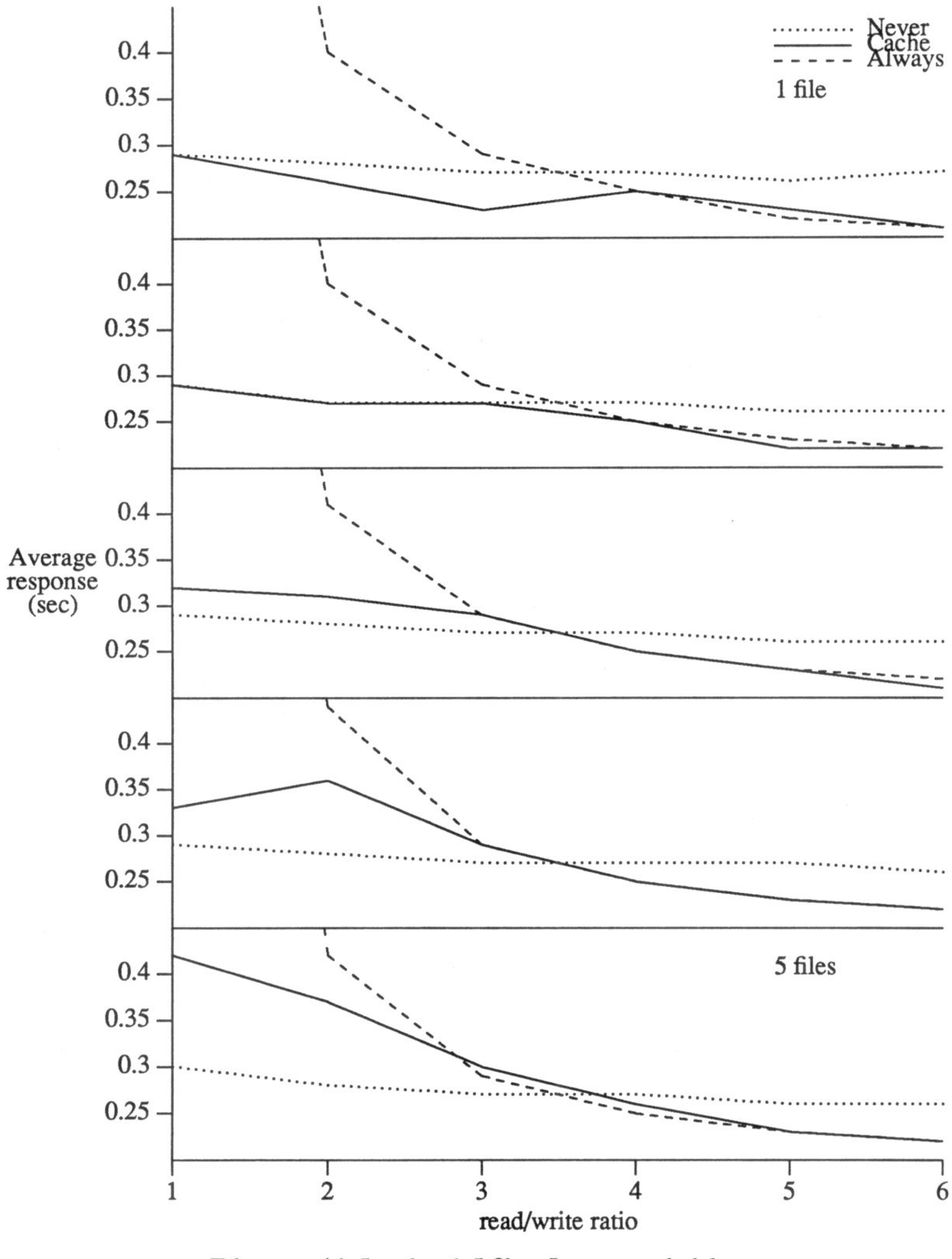
Binary tree with 7 nodes, 1-5 files, 20ms network delay

Figure 4.7



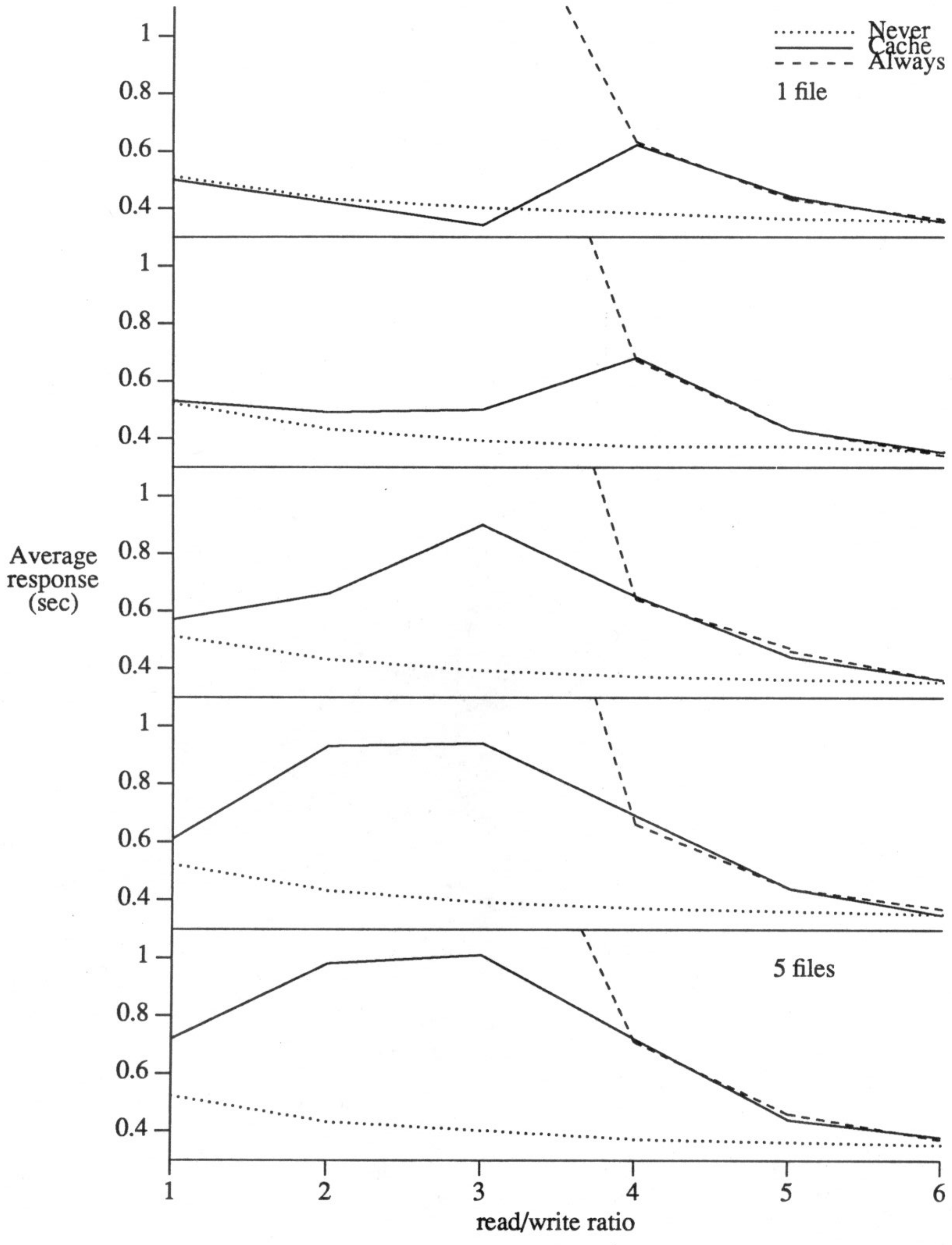
Binary tree with 7 nodes, 1-5 files, 40ms network delay

Figure 4.8



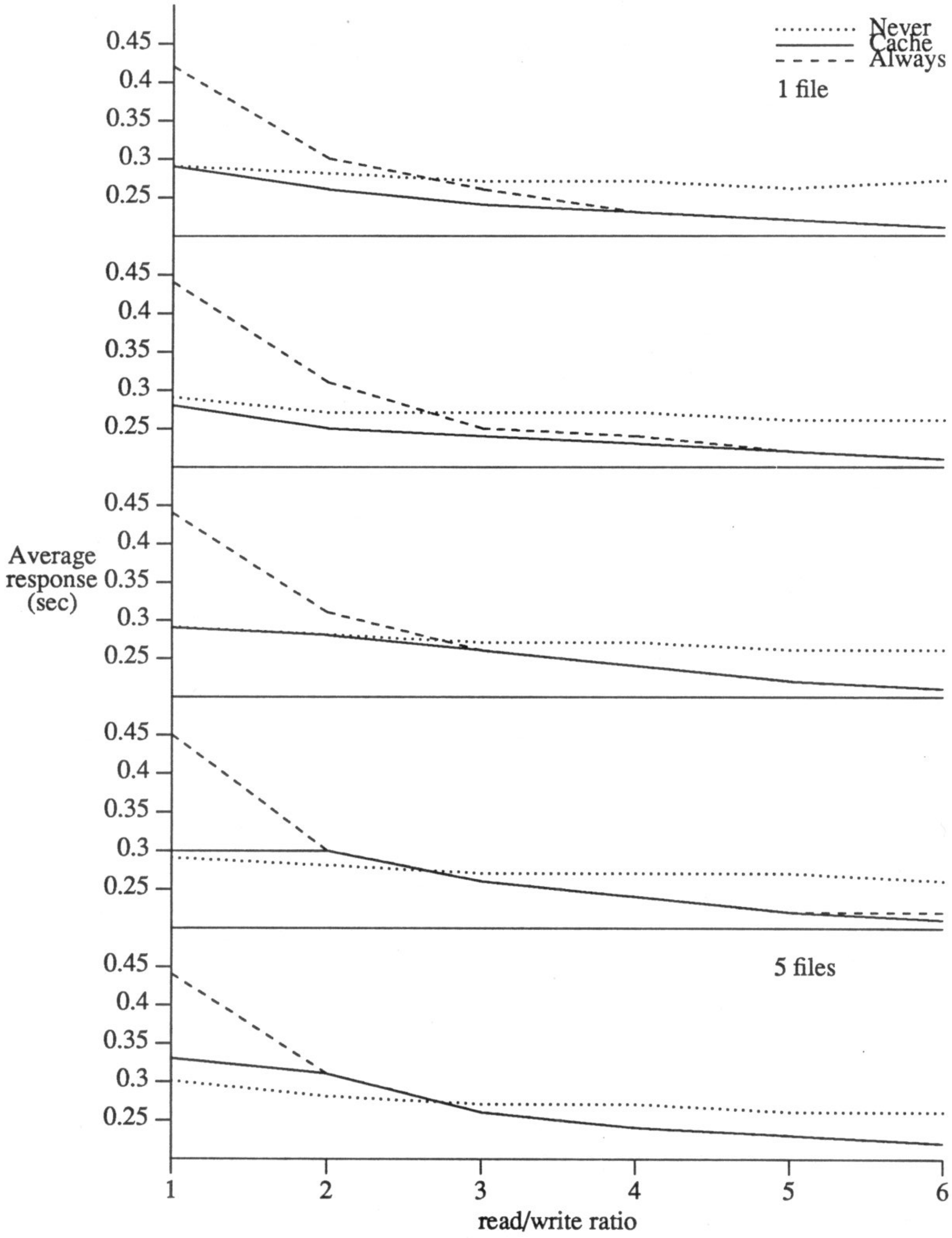
Ethernet with 5 nodes, 1-5 files, 5ms network delay

Figure 4.9



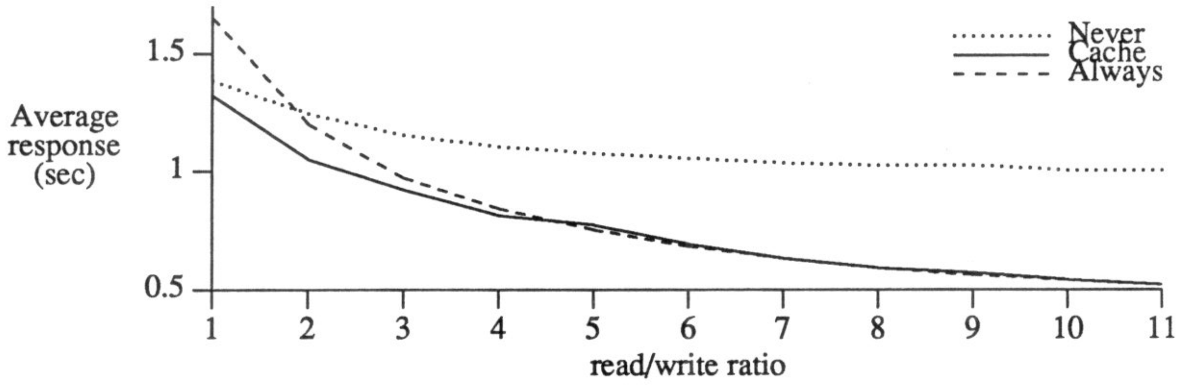
Ethernet with 5 nodes, 1-5 files, 10ms network delay

Figure 4.10



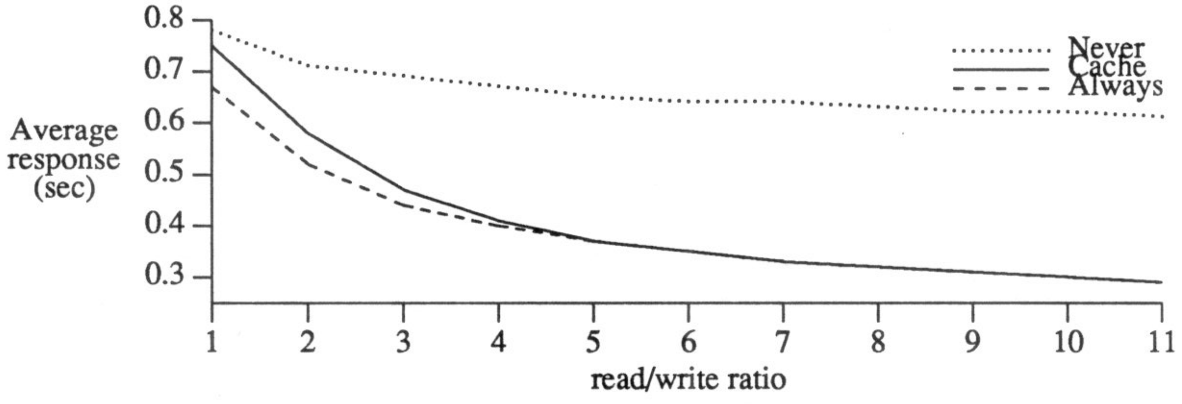
Ethernet with broadcasting: 5 nodes, 1-5 files, 5ms network delay

Figure 4.11



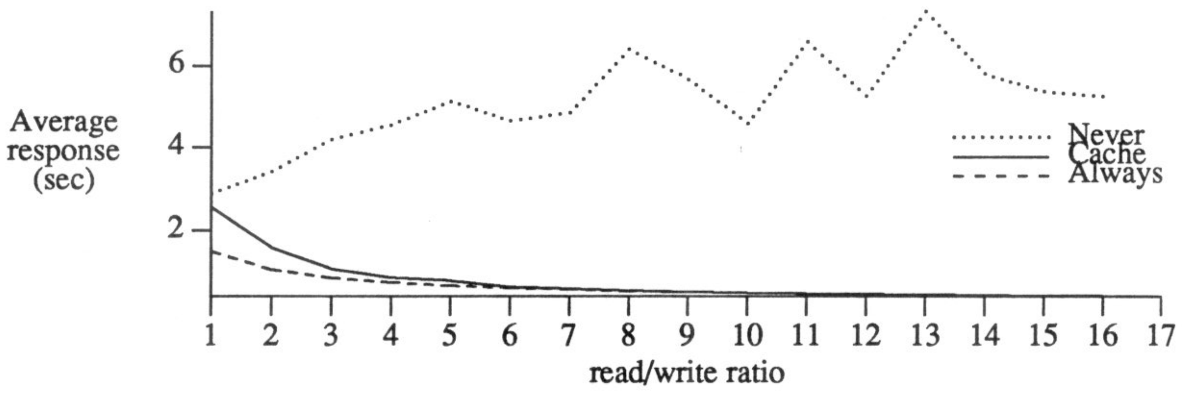
Ring configuration with 10 nodes, 5 files, 20ms network delay

Figure 4.12



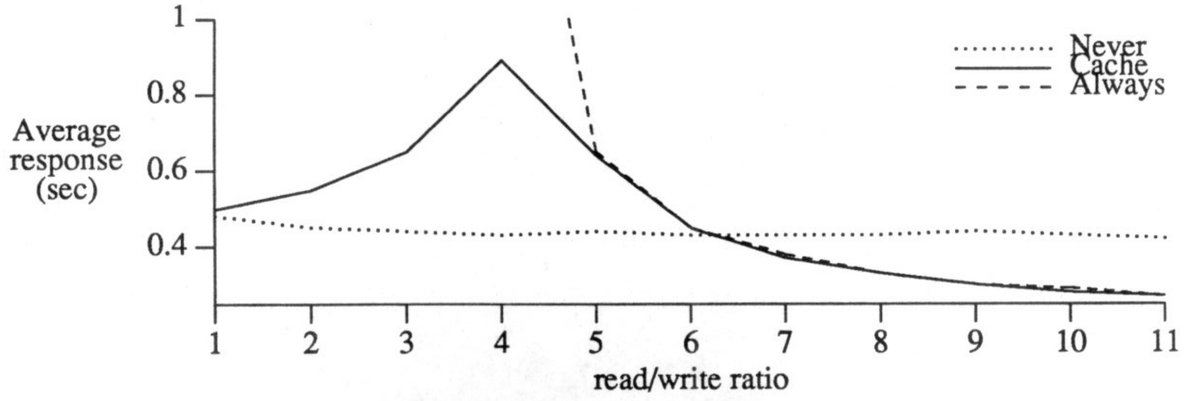
Fully connected network with 10 nodes, 5 files, 20ms network delay

Figure 4.13



Binary tree with 15 nodes, 5 files, 20ms network delay

Figure 4.14



Ethernet with broadcasting: 10 nodes, 5 files, 5ms network delay

Figure 4.15

4.6. Conclusions

Although our algorithm was developed for arbitrary networks, our results show that the design of a caching strategy must consider the topology of the network as well as the r/w ratio of the files. In a ring configuration, our caching strategy is the better choice. In a fully connected network, files should always be cached, and in an Ethernet, the best strategy never caches files at low r/w ratios and always caches them at higher r/w ratios.

The simulation results depend on the assumption that all files have the same, constant, r/w ratio. An important area for future work is to explore the significance of these constraints, and to examine what happens if they are relaxed.

It is simpler to relax the assumption that all files have the same r/w ratio; we can assign different r/w ratios to each file. Our analysis required only that each file have its own, fixed r/w ratio, so our strategy remains a valid caching policy. In this case, the mean and variance of the r/w ratio may be the factors that decide which of the three caching strategies is best for a particular network.

We can also relax the assumption that r/w ratios are constant by noting that variations in r/w ratios are not random, but cyclic. For example, a file in a banking application has one r/w ratio during the day when people are accessing accounts, and a different one at night when only batch jobs are running. We can divide the day into periods during which a file's r/w ratio is constant and design an appropriate strategy for each of these periods.

A fixed r/w ratio for each file is the most difficult assumption to relax. Our algorithm relies on this assumption and becomes invalid if the assumption is relaxed. However, the analysis in Section 4.3, up to the point where the assumption was made, is still valid, and the strategy of minimizing $\sum_{i}^{n} c_{ij}$ can, at least in theory, be used. However, the quantities involved are hard to obtain at every site, and this algorithm may not be applicable in practice without making other assumptions.

References

1. Lawrence W. Dowdy and Derrel V. Foster, ''Comparative Models of the File Assignment Problem,'' *ACM Computing Surveys*, vol. 14, no. 2, June 1982.
2. Rafael Alonso and Kriton Kyrimis, ''A Process Migration Implementation for a UNIX System.,'' in *Proceedings of the Winter 1988 Usenix Conference*, pp. 365-372, Dallas, Texas, February 1988.
3. Michael L. Powell and Barton P. Miller, ''Process Migration in DEMOS/MP,'' *Proceedings of the 9th ACM Symposium on Operating Systems Principles, Operating Systems Review*, vol. 17, no. 5, pp. 110-119, October 1983.
4. David A. Butterfield and Gerald J. Popek, ''Network Tasking in the Locus Distributed Unix System,'' in *Proceedings of the Summer 1984 Usenix Conference*, pp. 62-71, 1984.
5. Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton, ''Preemptable Remote Execution Facilities for the V-System,'' *Proceedings of the 10th ACM Symposium on Operating Systems Principles, Operating Systems Review*, vol. 19, no. 5, pp. 2-12, December 1985.
6. Fred Douglis, ''Process Migration in the Sprite Operating System,'' U. C. Berkeley Technical Report, 1987.
7. John Ousterhout, Andrew Cherenson, Fred Douglis, Michael Nelson, and Brent Welch, ''An Overview of the Sprite Project,'' *;login: The USENIX Association Newsletter*, vol. 12, no. 1, January/February 1987.
8. Mordecai B. Rosen and Michael J. Wilde, ''NFS Portability,'' in *Proceedings of the Summer 1986 Usenix Conference*, pp. 299-305, 1986.
9. S. R. Kleiman, ''Vnodes: An Architecture for Multiple File System Types in Sun UNIX,'' in *Proceedings of the Summer 1986 Usenix Conference*, pp. 238-247, 1986.
10. D. M. Ritchie and K. Thompson, ''The UNIX Time-Sharing System,'' *ACM Communications*, July 1974.
11. Brian W. Kernighan and Dennis M. Ritchie, *UNIX Programming–Second Edition,* Bell Laboratories, 1978.
12. *UNIX Commands Reference Manual,* Sun Microsystems Ltd., 1985.
13. *UNIX Interface Reference Manual,* Sun Microsystems Ltd., 1986.
14. K. Thompson, ''UNIX implementation,'' *Bell System Technical Journal*, vol. 57, no. 6, pp. 1931-1946, July-Aug. 1978.
15. Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry, ''A Fast File System for UNIX,'' *ACM Transactions on Computer Systems*, vol. 2, no. 3, pp. 181-197, August 1984.
16. Will E. Leland and Teunis J. Ott, ''Load-balancing Heuristics and Process Behavior,'' in *Proceedings of PERFORMANCE '86 and ACM SIGMETRICS 1986*, pp. 54-69, May 1986.
17. Luis-Felipe Cabrera, ''The Influence of Workload on Load Balancing Strategies,'' in *Proceedings of the Summer 1986 Usenix Conference*, pp. 446-458, Atlanta,

Georgia, 1986.

18. Derek L. Eager, Edward D. Lazowska, and John Zahorjan, "The Limited Performance Benefits of Migrating Active Processes for Load Sharing," Technical report 87-12-10, University of Washington, Department of Computer Science, December 1987.

19. Phillip Krueger and Miron Livny, "A Comparison of Preemptive and Non-Preemptive Load Distributing," in *Proceedings of the 8th International Conference on Distributed Computing Systems*, pp. 123-130, June 1988.

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

21. Matt W. Mutka and Miron Livny, "Profiling Workstations' Available Capacity for Remote Execution," Computer Sciences Technical Report #697, University of Wisconsin-Madison, May 1987.

22. Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon, "Design and Implementation of the Sun Network Filesystem," in *Proceedings of the Summer 1985 Usenix Conference*, pp. 119-130, 1985.

23. James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. Rosenthal, and F. Donelson Smith, "Andrew: A Distributed Personal Computing Environment," *Communications of the ACM*, vol. 29, no. 3, pp. 184-201, March 1986.

24. Michael Nelson, Brent Welch, and John Ousterhout, "Caching in the Sprite File System," Technical Report No. UCB/CSD 87/345, March 1987.

25. Amir Milo and Ouri Wolfson, "Placement of Replicated Items in Distributed Databases," *Proceedings of the International Conference on Extending Database Technology, EDTB '88*, Venice, Italy, March 1988.

26. Shojiro Muro, Toshihide Ibaraki, Hidehiro Miyajima, and Toshiharu Hasegawa, "Evaluation of the File Redundancy in Distributed Database Systems," *IEEE Transactions on Software Engineering*, vol. 11, no. 2, February 1985.

27. R. M. Bryant, "A Tutorial on Simulation Programming With SIMPAS," *Proceedings of the 1981 Winter Simulation Conference*, pp. 363-377, Atlanta, Georgia, December 9-11, 1981.