Performance of Optimistic Make

Rick Bubenik
Willy Zwaenepoel

Department of Computer Science
Rice University
Houston, Texas

Abstract

Optimistic make is a version of *make* that executes the commands necessary to bring targets up-to-date *prior* to the time the user types a make request. Side effects of these *optimistic computations* (such as file or screen updates) are concealed until the make request is issued. If the inputs read by the optimistic computations are identical to the inputs the computation would read at the time the make request is issued, the results of the optimistic computations are used immediately, resulting in improved response time. Otherwise, the necessary computations are reexecuted.

We have implemented optimistic make in the V-System on a collection of SUN-3 workstations. Statistics collected from this implementation are used to synthesize a workload for a discrete-event simulation and to validate its results. The simulation shows a speedup distribution over pessimistic make with a median of 1.72 and a mean of 8.28. The speedup distribution is strongly dependent on the ratio between the target out-of-date times and the command execution times. In particular, with faster machines the median of the speedup distribution grows to 5.1, and then decreases again. The extra machine resources used by optimistic make are well within the limit of available resources, given the large idle times observed in many workstation environments.

1 Introduction

Make is a tool used primarily for creating up-to-date executable programs from their source files [5]. Using a *makefile*, the user specifies a number of *targets*, the *sources* they depend on, and the commands to be executed to construct the targets from the sources. A target is said to be *out-of-date* if one of its sources has a larger timestamp than the target. When the user types **make**, out-of-date targets are reconstructed according to the makefile, possibly using multiple machines if some of the commands are independent.

Optimistic make is identical in functionality to make, but unlike the conventional "pessimistic" implementation, optimistic make monitors the file system for out-of-date targets, and executes the commands necessary to bring the targets up-to-date before the user types **make**. Outputs of these optimistic computations are concealed until the user types **make**. If the inputs used by the optimistic computations are unchanged when the make request is issued, their results are used immediately. Otherwise, the necessary computations are reexecuted.

Figure 1 shows the potential performance benefits of optimistic make over pessimistic make. The top portion of the figure depicts a pessimistic distributed make, whereby the user edits and saves a number of files, and then issues a make request, at which time the commands necessary to bring the targets up-to-date are executed. The bottom part of Figure 1 depicts the operation of optimistic distributed make. Commands are started as soon as files are saved, when targets become out-of-date. As a result, response time is significantly improved.

The outline of the rest of this paper is as follows. Section 2 briefly discusses the notion of ***encapsulations***, the basic construct used in the implementation of optimistic make. Section 3 describes the statistics collected from our implementation of optimistic make. In Section 4 we describe the simulation model used to evaluate the performance of optimistic make. Results from this simulation are presented in Section 5. Related work is covered in Section 6, and conclusions are drawn in Section 7.

This work was supported in part by the National Science Foundation under Grant CCR-8716914.

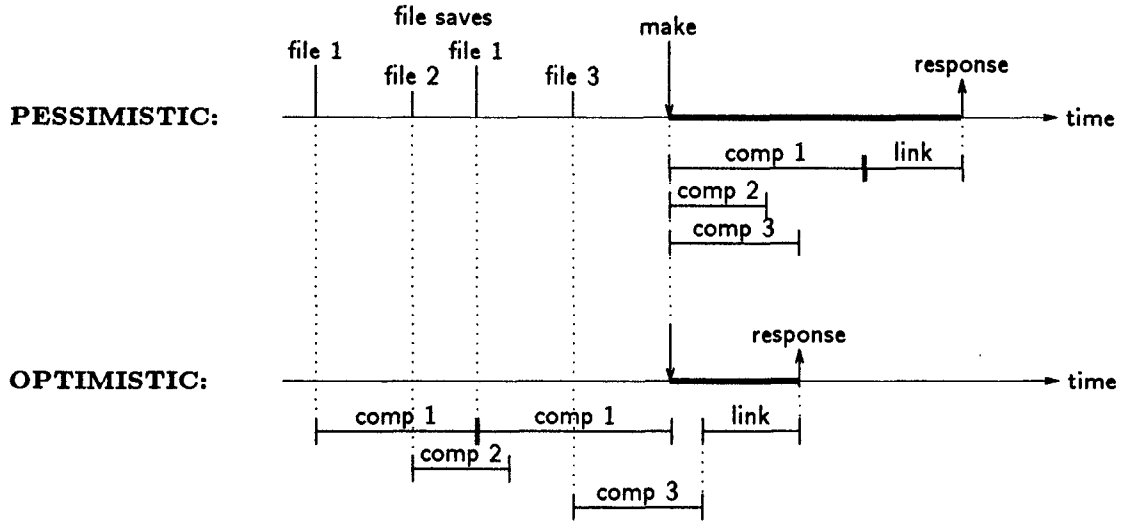

Figure 1 Optimistic vs. Pessimistic Distributed Make.

2 Encapsulations

Encapsulations are the primary mechanism used to support optimistic make. In this section, we summarize their functionality and those aspects of the implementation that are relevant to the performance of optimistic computations.

2.1 Definition

Informally, an encapsulation is a computation that runs with its outputs concealed until the computation is *mandated* (requested by the user). Outputs include, but are not limited to, file modifications and terminal output. The following three operations are defined on encapsulations:

eid = CreateEncapsulation() Create an encapsulation with unique identifier *eid*. Output produced by the encapsulation is not visible outside the encapsulation until it is mandated, with one exception: it is possible to allow one encapsulation to read the outputs of one or more *input encapsulations* by specifying these encapsulations as arguments to the **CreateEncapsulation()** call.

result = MandateEncapsulation(eid) Check whether the inputs read by the encapsulation are identical to the inputs that the computation would read if it were to run at this time. If so, then make all outputs produced so far visible, stop concealing further output, and return *success*. If not, abort the computation and return *failure*.

AbortEncapsulation(eid) Abort the encapsulation.

Encapsulations are superficially similar to atomic transactions in that both mechanisms hide operations until a later time (commit time for atomic transactions, mandate time for encapsulations). However, the semantics of encapsulations differ considerably from the semantics of transactions. Encapsulations can be mandated before the concealed computation completes. When mandated, side effects are made visible in steps (in the order in which they were created) rather than atomically. Encapsulations can be destroyed at any time, including while concealed side effects are being made visible. This lets the user abort unwanted computations before all (unwanted) output has appeared.

2.2 Implementation and Performance

Encapsulations are completely transparent to the computation. The same executable program can be run both as a normal computation and as an encapsulation. In particular, no recompilation or relinking of existing programs is necessary.

Unlike client programs, the kernel and server programs require modification to support encapsulations. The kernel tags each message from an encapsulation with the encapsulation identifier *eid*. This allows servers to determine efficiently whether a request comes from an encapsulation. A server must log the fact that an encapsulation reads one of its objects. This is done by logging a logical timestamp. Furthermore, when an encapsulation modifies an object, this modification must be redirected to a hidden object, not visible outside the encapsulation. On mandate, the server checks whether the timestamps of all objects read by the encapsulation are equal to their current timestamps. If so, the mandate is allowed to succeed, and all modifications

are made visible by replacing the original objects with the hidden ones produced during the encapsulation. If not, the encapsulation is aborted and all hidden objects are discarded.

In the case of the file server, each time an encapsulation opens a file for read, the file's timestamp is logged. When an encapsulation opens a file for write, a hidden file is created, and the writes are redirected to that file. Hidden files do not appear in the file system directory structure and are only accessible through low level identifiers. Writes to the terminal screen are also redirected to a hidden file. Furthermore, for each encapsulation, a hidden file system tree is maintained to record the modifications made by that encapsulation to the real file system tree. The hidden file system tree is also used to record the mapping between the names of files modified by the encapsulation and the low level identifiers of the corresponding hidden files.

The overhead of executing an encapsulation compared to a normal computation is (roughly) proportional to the number of *opens*, as opposed to the number of reads or writes. In our implementation, running under the V-System on SUN-3/50 workstations, the extra overhead is 18 milliseconds per open for read, and 8 milliseconds per open for write, for the first open of each file. The extra overhead is lower if the same file is opened again: 10 milliseconds per open for read and 4 milliseconds per open for write. The overhead is lower on subsequent opens since the hidden file system tree need not be updated. In the current implementation, encapsulations are provided by a separate *encapsulation server*. Much of the encapsulation overhead results from communication between the file server and the encapsulation server, and from the cost of maintaining the hidden file system tree and the logging. An implementation where the encapsulation server is integrated with the file server might be more efficient, but we prefer the modularity of our approach.

At mandate time, overhead is minimized by examining a number of timestamps in a single operation. We measured an overhead of 8 milliseconds per open for read, and 31 milliseconds per open for write. These times are limited by the time it takes our file server to lookup and to overwrite a file, respectively. When a computation is mandated while still executing, the mandate can proceed in parallel with the computation, so the overhead does not contribute to the computation's response time. For the types of computations considered in this paper (compilations and linkages), a conservative estimate for the encapsulation overhead is 2 seconds during execution and 1 second at mandate time.

2.3 Optimistic Make and Encapsulations

The optimistic make program reads the makefile and monitors the file system. File system monitoring is done efficiently by asking the file server for notification if any file in a specified directory changes. This results in shorter notification times and less overhead on the file server than polling, while keeping the amount of state to be maintained at the file server for this purpose small. When optimistic make sees a target in the makefile that is out-of-date, it starts an encapsulation to bring that target up-to-date. If two (or more) computations are necessary to bring a target up-to-date (for instance, a compilation and a linkage), the first computation is started as an encapsulation eid_1 without input encapsulations. When it finishes, the second computation is started as an encapsulation eid_2 with the first encapsulation eid_1 as an input encapsulation. This allows the linker to read the output of the compiler. If a source file changes after an encapsulation has been started, the corresponding encapsulation is aborted, and a new one is started. If any encapsulation in a sequence of dependent encapsulations is aborted, all subsequent encapsulations in the sequence are also aborted.

3 Measurements

3.1 Measurement Environment

The system used for measurement consists of from 8 to 12 diskless SUN-2/50 and SUN-3/50 workstations and a SUN-3/160 file server connected by a 10 megabit Ethernet. All machines are running the V-System [2]. Remote execution of programs is transparent and incurs only a very small performance penalty. File access is transparent as well, and has equal cost from all diskless machines. The status of other machines on the network can be obtained efficiently using the V group communication mechanism [3].

The machines are used for software development by our group, which consists of 8 graduate students and faculty members, and for projects in a graduate distributed systems course. Most of our makefiles involve C compilations and linkages, with a small number of Modula-2 compilations and some TEX text processing. There are typically 4 to 6 active users on the system during the day, although commonly only 2 or 3 of these are actually engaged in software development.

3.2 Method of Measurement

We have instrumented our make programs (both the pessimistic and optimistic versions) to collect the following statistics each time a make request is executed:

- The out-of-date time for all out-of-date targets: the difference between the current time and the largest

timestamp of any of the target's sources.

- Command execution time: the sequential execution time of each program executed as part of the make. All times are normalized to SUN-3's.
- The shape of the dependency graph and the number of computations executed as part of the make.
- The number of encapsulations aborted as part of each optimistic make.

Statistics were gathered for more than 6 months. Approximately 4,000 requests were measured over this period.

3.3 Measurement Results

Figure 2 shows the cumulative distribution of the target out-of-date times. This distribution shows a median value of 32 seconds and a mean value of 378 seconds. This implies that most targets are requested fairly soon after a change to the source files is made. Occasionally, however, users wait quite a while before executing a make request. Figure 3 shows the cumulative distribution of the command execution times. The distribution varies with the number of commands per make request, where requests with a small number of commands have lower command execution times. We speculate that this is due to the fact that make requests with a small number of commands (and especially those with one command) frequently contain computations aborted due to compilation errors.

Virtually all makefiles have a similar dependency graph (see Figure 4): a number of independent computations (usually compilations) followed by a single computation (usually a linkage). The distribution of the number of commands per make is given in Figure 5. The median number of commands per make request is 2 (corresponding to a change to a single source file, resulting in a recompilation of that source file and a linkage). The mean number of commands is 4.39.

3.4 Overhead Estimates

Optimistic make uses more CPU resources due to the aborted optimistic computations and the encapsulation overhead. Table 1 shows the measured number of computations mandated and aborted with optimistic make. For each necessary computation (i.e., each computation that would also be necessary in pessimistic make), 1.39 optimistic computations are started on average. Hence, aborted computations impose an average extra CPU load of at most 39 percent. This is an upper limit on the extra load since many of the aborted computations do not run to completion, and thus use less CPU time. For the computations considered here (compilations and

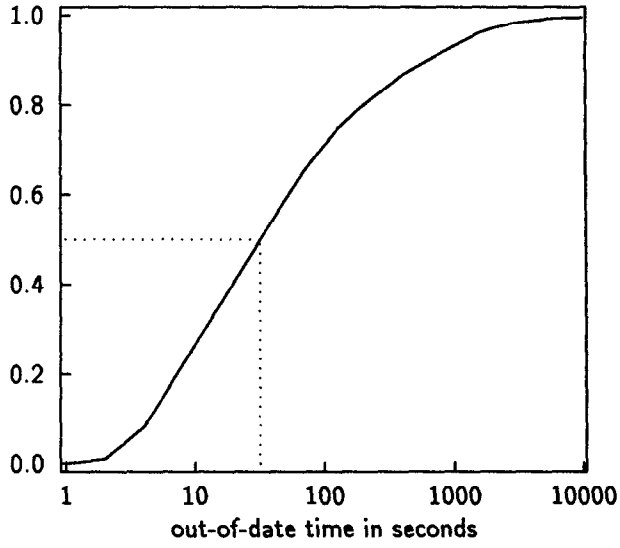

Figure 2 Cumulative Distribution of Target Out-of-date Times.

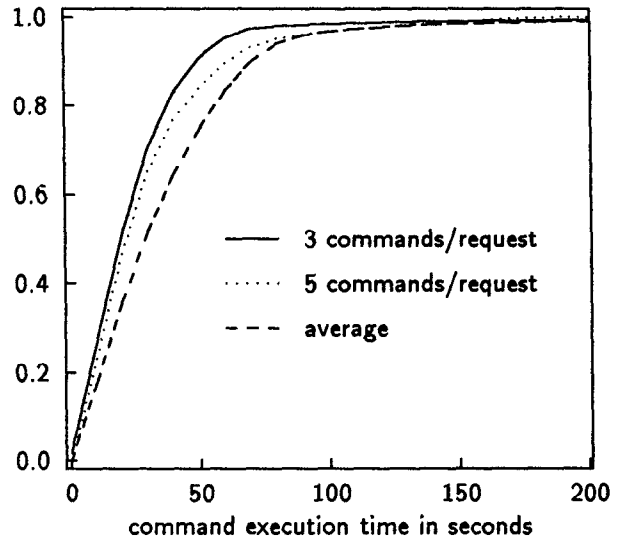

Figure 3 Cumulative Distribution of Command Execution Times.

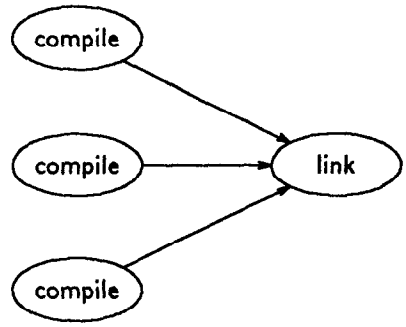

Figure 4 Typical Makefile Dependency Structure.

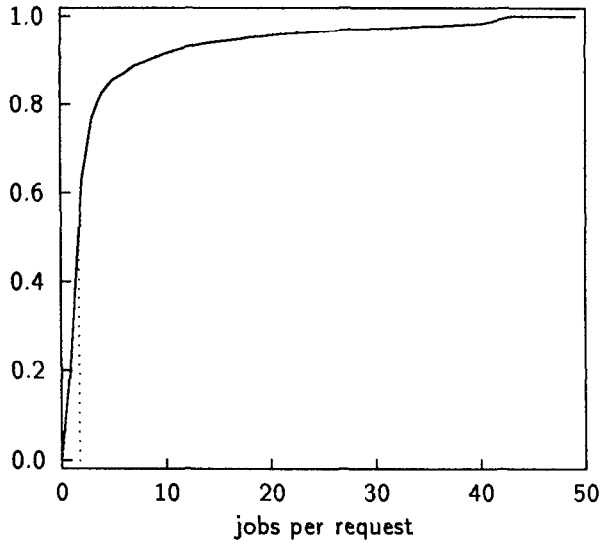

Figure 5 Cumulative Distribution of Number of Computations per Request.

Computations	Number	Percent
necessary	16634	100%
aborted	6448	39%
total	23082	139%

Table 1 Necessary and Aborted Computations.

linkages), encapsulation overhead adds less than 5 percent overhead on average (see Section 2.2). Hence, we conclude that the total extra load is at most 44 percent, and in practice is significantly lower. This extra CPU load is small compared to the very large idle times that have been observed in workstation environments, even during peak usage periods [10].

Encapsulations use extra disk space beyond that used by normal computations to store the hidden files. To estimate an upper bound of how much extra space might be used, we assume that each user has a completed optimistic make containing the measured average of 4.39 computations. These normally consist of a link (producing an executable file) and an average of 3.39 compilations (producing object modules). Using the average executable and object module sizes in our system, each of these optimistic makes requires a total of 81 kilobytes. If we assume the typical file server has at least 10 megabytes per client, this represents less than 1 percent of the client's disk allocation.

4 Simulation

The simulation model consists of N identical machines and M users. Each user issues make requests, with the think time between requests drawn from an exponential distribution. A computation may use any of the machines, although at any time we only allow a single computation to execute on a particular machine. A centralized allocator assigns computations to machines in a FCFS order, preferring normal to optimistic computations. In practice, the machine where a make request is issued chooses the machines to be used for execution of the commands belonging to this request. While different from centralized allocation, the individual machines in our environment have sufficiently accurate information about the status of other machines for centralized allocation to be a reasonable approximation. Once a computation is started, it runs to completion (unless aborted), with no preemption. When all workstations are busy, requests are queued until a computation completes.

We simulate both pessimistic and optimistic make with identical arrivals of make requests. For each pessimistic make request, we draw the number of commands to be executed from the empirical distribution shown in Figure 5, and then select the command execution times from the distribution in Figure 3 for requests with that number of commands. The commands are started when the pessimistic make request arrives, subject to the dependencies in the makefile. Only dependencies of the form depicted in Figure 4 are considered. For optimistic make, we draw the command execution times and number of commands from the same distributions as for pessimistic make, and additionally we draw the out-of-date times for each of the sources from the empirical distribution shown in Figure 2. The commands for the optimistic make are started at the time of the make minus the time drawn from the out-of-date time distribution. In order to simulate aborted computations in optimistic make, we introduce an extra command for P percent of the optimistic commands, where P is normally set to the measured 39 percent. We assume both pessimistic and optimistic make have negligible request processing overhead. In order to account for encapsulation overhead, optimistic computations are assessed an extra overhead of 2 seconds during execution and 1 second at mandate time.

The purpose of the simulation is to determine the *response time improvement* of optimistic make over pessimistic distributed make. *Response time* is the difference between the time the make request is issued and the time all commands corresponding to that make request are completed. *Response time improvement* is the ratio of response time in pessimistic make over response time in optimistic make. Since the improvement is dependent on the particular make request and the out-of-date times, we provide as the main result of our simulations the cumulative distribution of the improvement

of optimistic over pessimistic make. For each simulated request, the response time improvement is computed, and the distribution of improvements is computed from these values. Additionally, we provide the median response times for both optimistic and pessimistic make as an indication of the absolute difference in response times.

We run a terminating (finite horizon) simulation for a period of 10 simulated hours. Pessimistic and optimistic results are compared by constructing a confidence interval on the median response time improvement for each run with a 95 percent approximate confidence and a relative precision of ±3 percent.[1]

5 Simulation Results

5.1 The Baseline System

Figure 6 shows a cumulative distribution for the response time improvement in a system configured similar to the one we are using. All simulation inputs are drawn from the empirical distributions, the number of machines was set to 10, the number users to 2, and the mean think time to 6 minutes.[2] The median response time improvement is 1.72, and the mean is 8.28. This reflects the fact that most make requests are issued relatively shortly after changes to the source files are made. Improvements are occasionally very high, when all optimistic computations have completed by the time of the make request, and the response time for the optimistic make is equal to the time necessary to mandate the computations.

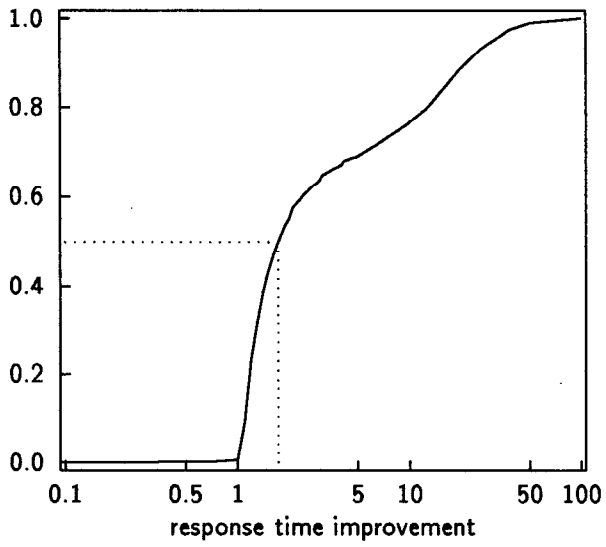

Figure 6 Cumulative Distribution of Response Time Improvement (Baseline System).

To validate the simulation model, we compare the measured cumulative response time distribution to the one obtained from the simulation, for both optimistic and pessimistic make (see Figure 7). We did not compare response time improvements since the improvement, as it is computed in the simulator, cannot be measured because a make request comes from either pessimistic or optimistic make, and not from both (as in the simulator). Hence, it is not possible to use the response time improvement distribution for the purpose of validation.

Response time improvement is affected mainly by the *ratio* of target out-of-date times to command execution times, and by the number of machines available for execution. The ratio of target out-of-date times to command execution times is important because it determines the amount of optimistic computation that can be executed before requested. To examine the effect of changing this ratio, we initially set the number of machines to infinity, then alternately vary the command execution and out-of-date times (Sections 5.2 and 5.3). In Section 5.4, we compare the machine utilization of pessimistic and optimistic make, then address the effect of limited machines in Section 5.5. Finally, the effects

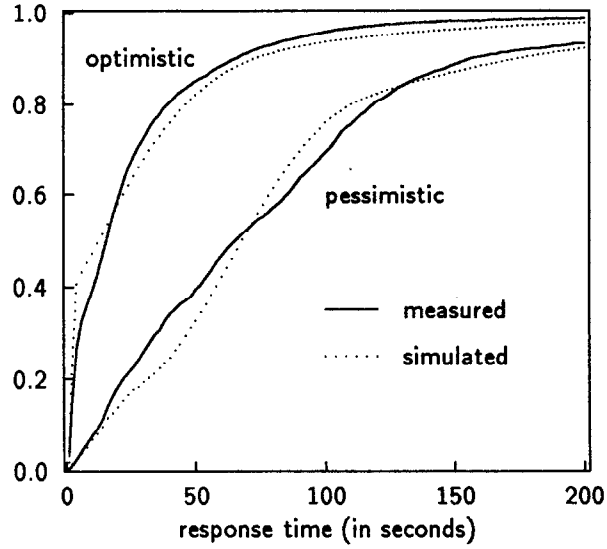

Figure 7 Cumulative Distributions of Simulated and Measured Response Times.

[1] In those experiments where the median pessimistic and optimistic response times are also recorded, the relative precision for all three statistics is set at ±3 percent, resulting in a lower aggregate precision.

[2] Preliminary measurements indicate the mean think time in our environment is at least 6 minutes. Consequently, we use this value for all simulations. We change the number of users and machines to experiment with increased system loading.

of scheduling algorithms are discussed briefly in Section 5.6.

5.2 Increasing Machine Speed

To assess the effect of shorter command execution times (for instance, as a result of faster machines), the number of machines is set to infinity, and the command execution times (from Figure 3) are divided by the appropriate factor.[3] Encapsulation overhead is also reduced accordingly. Other inputs to the simulation (out-of-date times and number of commands per make) are as in the baseline model.

Figure 8 shows the cumulative distribution of response time improvement for the original machine speed (labeled SUN-3), and for systems 8 and 16 times faster. Figure 9 shows the median response times for pessimistic and optimistic make plotted side-by-side for several CPU speeds.[4] Figure 9 shows that as machine speed increases, the absolute difference between optimistic and pessimistic make decreases. The response time improvement, however, first grows and then decreases with faster machines, from a median of 1.7 in the SUN-3 curve, to a maximum of 5.1 in the 8*SUN-3 curve, and then back down to a median of 3.3 in the 16*SUN-3 curve. As the machine speed goes from

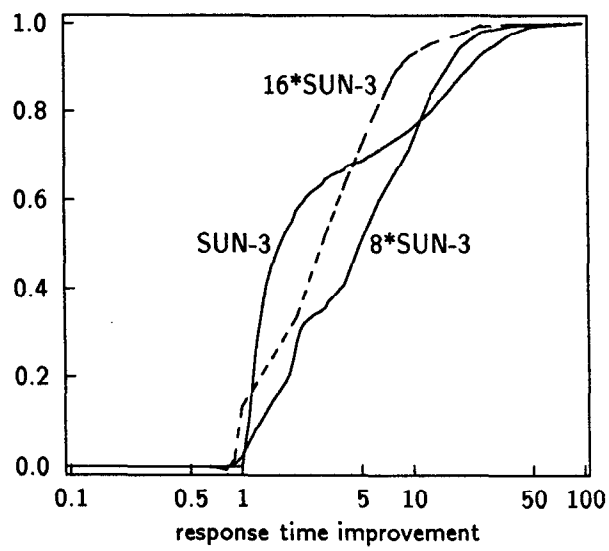

Figure 8 Cumulative Distribution of Response Time Improvement for Different Machine Speeds.

[3]The number of users and the think time are irrelevant with an infinite number of machines.

[4]The ratio of the median response times is not the same statistic as the median response time ratio (the latter is computed by selecting the median of all individual improvements).

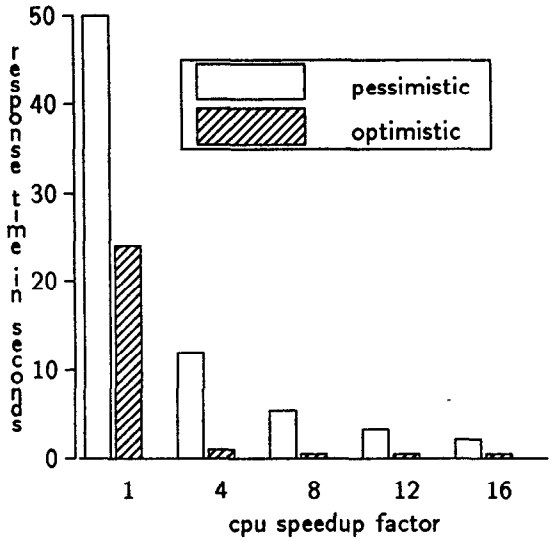

Figure 9 Median Response Times for Different Machine Speeds.

SUN-3 to 8*SUN-3, a large number of optimistic computations are completed or are near completion by the time the make request is issued. Hence, response time for optimistic make is drastically improved. Response time for pessimistic make improves as well, but not as fast, giving rise to a better response time improvement. Beyond the point where almost all optimistic computations are completed by the time of the make request, there is little improvement in optimistic make's response time as a result of faster machines. Pessimistic make, however, continues to improve, leading to a decreasing response time improvement.

5.3 Changing Out-of-Date Times

When collecting statistics, we observed that the median and mean of the out-of-date time distribution fluctuated somewhat over different measurement periods. We simulate varying out-of-date times by using values drawn from the empirical out-of-date time distribution multiplied by a scale factor of X. Other simulation inputs are as in the baseline system, with an infinite number of machines.

Figure 10 shows the cumulative distributions for factors of 0.25, 1, and 4. Figure 11 shows the median response times for pessimistic and optimistic make plotted side-by-side for several scale factors between 0.25 and 8. Unlike with increasing machine speed (Figures 8 and 9), larger out-of-date times increase both the response time improvement and the absolute difference between median response times, until virtually all optimistic computations are completed by mandate time. With even larger out-of-date times, both remain constant, again in contrast with Section 5.2, where faster

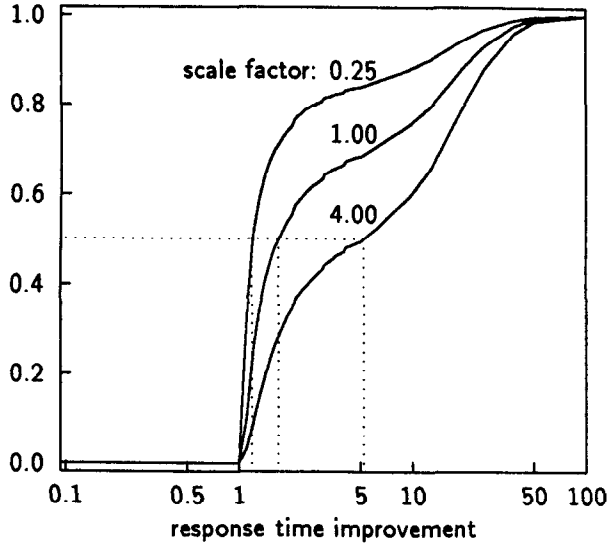

Figure 10 Cumulative Distribution of Response Time Improvement for Different Out-of-date Scale Factors.

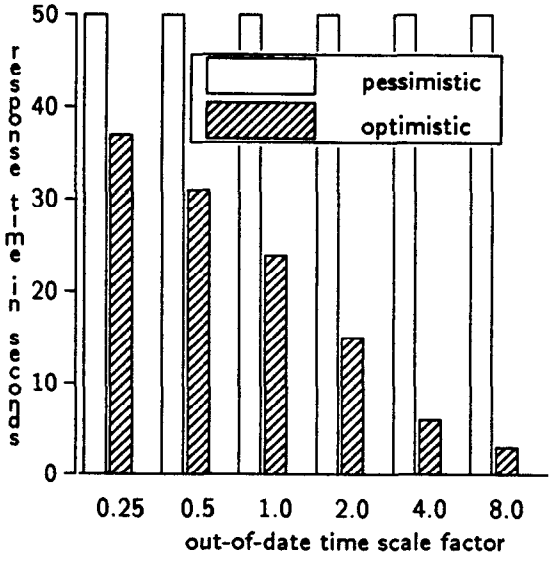

Figure 11 Median Pessimistic and Optimistic Response Times for Different Out-of-date Scale Factors.

machines cause pessimistic make's response times and hence the response time improvement to decrease.

5.4 Machine Utilization

Figure 12 shows the probability distribution for the number of busy machines with optimistic make using 39 percent aborted computations (the percentage measured). This distribution is obtained by sampling the number of busy machines at 1 minute intervals during the simulation. In these simulations, the number of users is varied between 1 and 16, while the think time is kept constant at 6 minutes.[5] Inputs for the simulations are drawn from the empirical distributions, and an infinite number of machines are available.

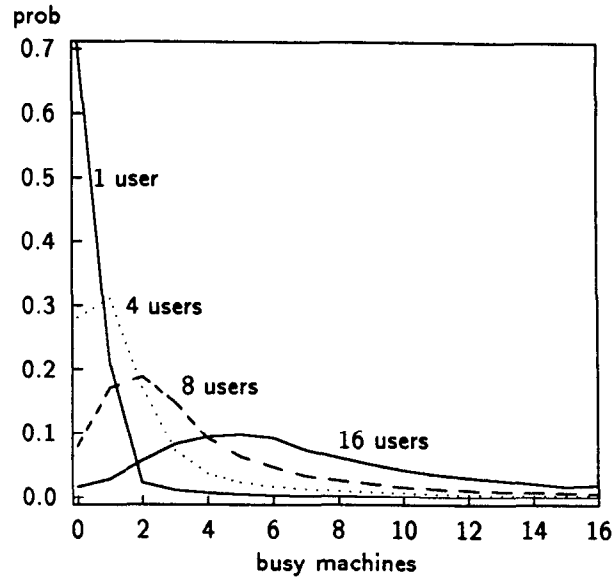

Figure 12 Probability Distribution of Busy Machines for Different Numbers of Users.

Figure 13 shows the probability distribution of the number of busy machines for 16 users for pessimistic make, optimistic make with no aborted computations, optimistic make with the measured 39 percent aborted computations, and optimistic make with 72 percent aborted computations (where all source node computations in the makefile dependency graph are aborted once). Figure 13 shows that optimistic make distributes CPU load more evenly over time: it is less likely to use very few machines or very many machines. This arises because pessimistic make needs many machines when the make request arrives, while optimistic make spreads out machine usage for each request by using machines as soon as files are saved. The aborted computations add to the overall machine utilization of optimistic make, but CPU usage remains less variable.

5.5 Limiting the Number of Machines

We now limit the number of machines, while fixing the number of users at 16 and the think time at 6 minutes. All other simulation inputs are taken from the empirical distributions. Figure 14 shows the speedup

[5]Simulations with a constant number of users and varying think times give similar results to those presented here for constant think time and varying numbers of users.

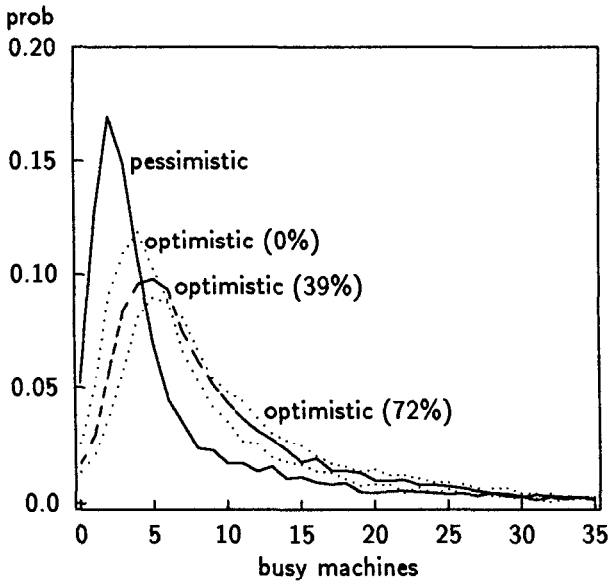

Figure 13 Probability Distribution of Busy Machines for Different Percentages of Aborted Computations.

distribution with 8, 16, and an infinite number of machines. Figure 15 shows the median response times for optimistic and pessimistic make for the same numbers of machines using the three abort ratios from above.

In going from an infinite number of machines to 16, the improvement remains approximately constant, since neither optimistic nor pessimistic make are machine limited in these circumstances. When further decreasing

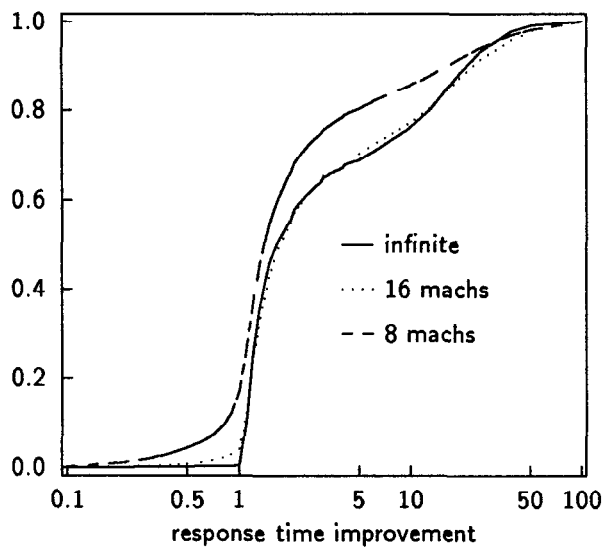

Figure 14 Cumulative Distributions of Response Time Improvement for Different Numbers of Machines.

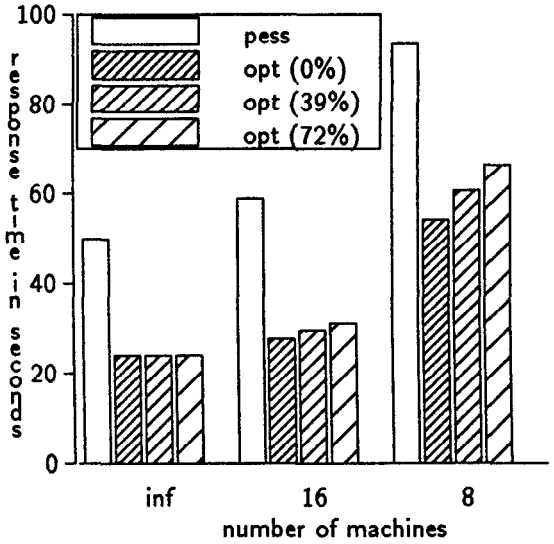

Figure 15 Median Response Times for Different Numbers of Machines.

the number of machines to 8 (with 2 users per machine), the improvement declines because optimistic computations are frequently blocked while requested computations use all the resources. The roughly constant improvement down to 16 machines (one user per machine) indicates optimistic make provides significant benefits under normal circumstances. Even with unexpectedly high loads, optimistic make still provides some improvement.

5.6 Effects of Scheduling Algorithms

In this section, we summarize preliminary results obtained from experimenting with different scheduling algorithms. The four scheduling algorithms we consider are 1) FCFS, 2) FCFS preferring normal commands to optimistic ones (the strategy used so far), 3) FCFS aborting optimistic commands if the machine is needed for a normal command, and 4) FCFS suspending optimistic commands if the machine is needed for a normal command. All four algorithms assume a centralized scheduler or distributed knowledge of what machines contain what commands. The simulation results indicate that the choice of algorithm makes no noticeable difference until the system load becomes abnormally high (for example, a fully loaded system with 1 minute think times). At this extreme, strict FCFS performs slightly worse than the other three algorithms, but each of these three perform essentially the same. Although further work is necessary, it appears that in our environment the choice of scheduling algorithm has relatively little impact on the improvement of optimistic make.

6 Related Work

Optimistic computations have been incorporated into the Integral C programming environment developed at Tektronix [11]. Unlike our implementation, which allows optimistic execution of arbitrary programs, their system only allows a small set of tools to be executed optimistically. No performance evaluation is given. There is also no evidence that Integral C conceals the output of optimistic computations, something we consider essential.

The eager evaluation work on functional and applicative programming languages is, to a lesser extent, related to our work [1, 7, 8, 9]. In these settings, with call-by-need semantics, arguments to functions are evaluated before they are known to be needed. The functional nature of the arguments obviates the need for explicit concealment of side effects. Our work is different in that we explicitly deal with side effects, and in that the grain of computation we consider is much larger. We believe that with a large grain of computation, the potential for optimistic computations increases significantly, since the overhead involved in concealing side effects becomes relatively less important.

There are interesting similarities and differences between our work and much of the work in load sharing [4, 6]. Load sharing attempts to improve throughput by spreading the workload equally over different machines. Optimistic execution attempts to decrease response time by spreading out the workload over time.

7 Conclusion

Optimistic make offers significant response time improvement under a wide variety of circumstances. The probability distribution of the response time improvement typically peaks early and then has a long tail, reflected in a small median and a large mean. In our current environment, the median improvement is around 1.7 and the mean improvement around 8. If, as expected, faster machines become available, the median improvement will grow significantly, until all optimistic computations are completed by the time the user types **make**. The amount of extra CPU and disk use resulting from optimistic make is limited. Given the increased availability of machines and the observed large idle time percentages in many workstation environments, the extra CPU utilization does not adversely affect performance.

References

[1] F.W. Burton. Controlling speculative computation in a parallel functional programming language. In *Proceedings of the Fifth International Conference on Distributed Computing Systems*, pages 453–458, May 1985.

[2] D. R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[3] D. R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[4] D.L. Eager, E.D. Lazowska, and J. Zahorjan. Adaptive load balancing in homogenous distributed systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.

[5] S. Feldman. Make—a computer program for maintaining computer programs. *Software Practice and Experience*, 9(4):255–265, April 1979.

[6] R. Hagmann. Process server: Sharing processing power in a workstation environment. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, pages 260–267, May 1986.

[7] R. H. Halstead. Parallel symbolic computing. *IEEE Computer*, 19(8):35–43, August 1986.

[8] D. A. Hornig. *Automatic Partitioning and Scheduling on a Network of Personal Computers.* PhD thesis, Carnegie-Mellon University, November 1984.

[9] P. Hudak and L. Smith. Para-functional programming: A paradigm for programming multiprocessor systems. In *Proceedings of the Thirteenth Annual Symposium on Principles of Programming Languages*, pages 243–254, January 1986.

[10] M. W. Mutka and M. Livny. Scheduling remote processing capacity in a workstation-processor bank network. In *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pages 2–9, September 1987.

[11] G. Ross. A practical environment for C programming. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 42–48, January 1987. Also available as SIGPLAN Notices 22(1), January 1987.