Optimistic Make

Rick Bubenik, *Member, IEEE*, and Willy Zwaenepoel, *Member, IEEE*

***Abstract*— *Optimistic make* is a version of the *make* program that begins execution of the commands needed to update *makefile* targets before the user issues the *make* request. Outputs of these *optimistic computations* (such as file or screen updates) are concealed until the request is issued. If the inputs read by the optimistic computations have not been changed by the time of the *make* request, the results of the optimistic computations are used, leading to improved response time. Otherwise, the necessary computations are reexecuted.**

We introduce the notion of *encapsulations* as the basic construct used to support optimistic *make*, and we describe the implementation of optimistic *make* in the V-System on a collection of SUN workstations. Statistics measured from this implementation are used to synthesize a workload for a discrete-event simulation, and to validate the simulation's 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 The speedup distribution is strongly dependent on the ratio between the target out-of-date times and the command execution times. With faster machines the median of the speedup distribution grows to 5.1, and then decreases again. Given the large idle times observed in many workstation environments, the extra machine resources used by optimistic *make* are well within the limit of available resources.

***Index Terms*— Encapsulation, optimistic computation, optimistic *make*, performance evaluation, programming environment, simulation study, software development environment, speculative computation.**

I. Introduction

MAKE is a tool used primarily in software development environments for creating up-to-date executable programs from their source files [9]. Using a *makefile*, the user specifies a number of *targets*, the *sources* that they depend on, and the commands necessary to construct the targets from the sources. A target is said to be *out-of-date* if one of its sources has a later timestamp than the target. When the user types *make*, out-of-date targets are reconstructed according to the *makefile*. If some of the commands are independent, they may be executed in parallel on separate machines.

Optimistic *make* is identical in functionality to *make*. However, unlike the conventional *pessimistic* implementation of *make*, it monitors the file system for out-of-date targets, executes the commands necessary to bring the targets up-to-date *before* the *make* request is issued, and conceals the outputs of the optimistically executed commands until the user types *make*. If the inputs read by the optimistic commands remain unchanged until the *make* request is issued, these optimistic results are used immediately. Otherwise, the necessary commands are reexecuted.

The operational differences between optimistic *make* and pessimistic *make*, and the potential performance benefits of optimistic *make* are shown in Fig. 1.

The top portion of the figure depicts the operation of 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 portion of the figure depicts the operation of optimistic distributed *make*. Commands are started as soon as files are saved, when targets become out-of-date. The response time for the *make* request is significantly improved since most command execution occurs before the request is issued.

The outline of the rest of this paper is as follows. Section II discusses the notion of *encapsulations*, the primary mechanism used to support optimistic *make*. Section III discusses the implementation of encapsulations and optimistic *make* in the V-System. Section IV presents the statistics collected from our implementations of both pessimistic *make* and optimistic *make*. Section V describes the simulation model used to further evaluate the performance of optimistic *make*. Results from this simulation are presented in Section VI. Related work is covered in Section VII. Finally, conclusions are drawn and further work is discussed in Section VIII.

II. Encapsulations

A. Definition

An *encapsulation* is a computation whose outputs are concealed until the computation is *mandated*. Once mandated, the outputs are made visible in an order consistent with the order in which they were produced during the execution of the encapsulation. 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 an encapsulation to read the outputs of one or more *input encapsulations*, before they are mandated, by specifying these encapsulations as arguments to the **CreateEncapsulation**() call. The newly created encapsulation is then said to be *dependent* on its input encapsulations.

result = *MandateEncapsulation(eid)*: If the inputs read by the encapsulation are unchanged, then reveal all outputs pro-

Manuscript received August 15, 1989; revised November 4, 1990. This work was supported in part by the National Science Foundation under Grants CDA-8 619893 and CCR-8716914 and by IBM Corporation under Research Agreement 16140046.

R. Bubenik is with the Department of Computer Science, Washington University, St. Louis, MO 63130.

W. Zwaenepoel is with the Department of Computer Science, Rice University, Houston, TX 77251.

IEEE Log Number 9103030.

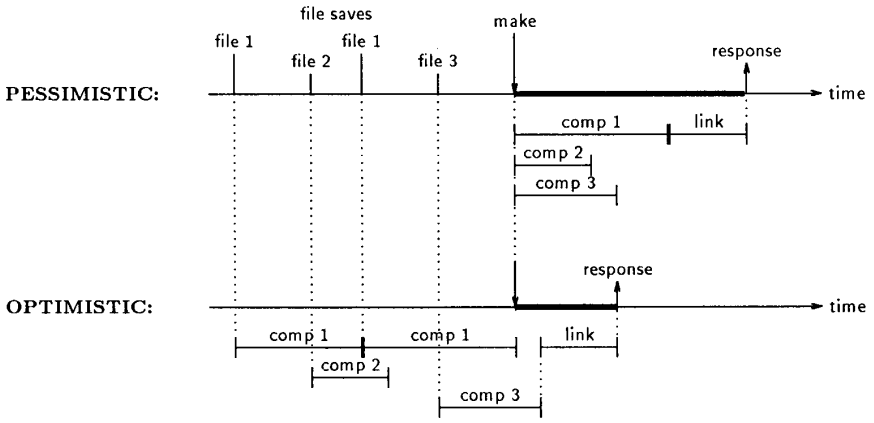


Fig. 1. Optimistic versus pessimistic distributed *make*.

duced so far, do not conceal further output, and return *success*. Otherwise, abort the encapsulation and return *failure*.

AbortEncapsulation(eid): Abort the encapsulation and discard its concealed output.

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 those of transactions. Encapsulations can be mandated before the concealed computation completes, allowing an encapsulation to be converted into a normal computation at any point during execution. When an encapsulation is mandated, output is made visible in steps rather than atomically. This simplifies the implementation by avoiding atomicity concerns. Encapsulations may be destroyed at any time, even while concealed output is being made visible, allowing the user to abort unwanted computations before the remaining unwanted output has appeared.

An extended version of encapsulations appears to be a useful abstraction for supporting *output commitment* in any optimistic computation [2], not only in optimistic *make*. In optimistic computations, all output must be concealed until the guess on which the optimistic computation is based can be confirmed. For instance, optimistic rollback recovery methods [14], [19] require that outputs be concealed until it is guaranteed that the states from which these outputs are performed will never be rolled back as a result of a failure. In this paper, we concentrate only on the use of encapsulations to support optimistic *make*.

B. Optimistic Make and Encapsulations

The optimistic *make* program reads an unmodified *makefile*, and monitors the file system for modifications to the source files on which the *makefile* targets depend. By default, the first target appearing in the *makefile* is the only target optimistically remade. This can be expanded or restricted by command line options. File system monitoring is done efficiently by requesting notification from the file server when any file in a specified list of directories is modified. This results in shorter notification times and less overhead on the file server than polling, while keeping small the amount of file server state to be maintained for this purpose. When optimistic *make* detects a target in the *makefile* that is out-of-date, it starts an encapsulation to update that target. If two (or more) dependent computations are necessary to update a target (for instance, a compilation followed by a linkage), the first computation is started as an encapsulation eid_1 without input encapsulations, and when it finishes, the second computation is started as an encapsulation eid_2 with the first encapsulation eid_1 as an input encapsulation. In the example of a compilation followed by a linkage, 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. When more than one independent encapsulation is necessary to update a target, the independent encapsulations are started concurrently on separate machines, assuming enough machines are available. If an encapsulation is not mandated within a certain timeout interval, the encapsulation may be aborted to release resources. The computation is then repeated if the user eventually issues the *make* request.

III. Implementation

This section describes an implementation of encapsulations and optimistic *make* in the V-System [4]. In this implementation, encapsulations are transparent to application programs. The same executable can be run either as a normal computation or as an encapsulation, with no need for recompilation or relinking.

The V-System follows the client–server model of application programs executing as client processes and accessing most operating system services by sending messages to server processes. The V kernel is a small kernel that provides efficient, location-independent message passing, and very little else.

We have adhered to the small-kernel philosophy in the implementation of encapsulations. An *encapsulation server*

process provides most of the support for encapsulations. Only minor modifications to the kernel and to some of the servers are required. However, not all servers need to be modified. Several encapsulation servers may be running at the same time, but a particular encapsulation and all its dependent encapsulations must be handled by the same encapsulation server.

A. Kernel Support

Two new fields are added to each kernel process descriptor record in order to support encapsulations: the *eid*, which contains the encapsulation identifier, and the *encapsulation flag*, which indicates whether this process supports encapsulations. The *eid* is zero, by default, for processes that do not run as encapsulations. A normal computation is converted into an encapsulation by instructing the kernel to set the *eid* to a specified nonzero value for all processes of the computation. The *eid* is also inherited by all processes created by the encapsulation. Servers that support encapsulations instruct the kernel to set the *encapsulation flag* in their process descriptor.

All messages are tagged with the *eid* of the sender. The kernel delivers messages sent by an encapsulation (that is, messages with a nonzero *eid* tag) only to other processes in the same encapsulation or to server processes that have indicated that they support encapsulations. Delivery of other messages sent by encapsulations is blocked until the computation is mandated, typically blocking the progress of the sending process as well. This allows encapsulations to be used in environments where not all servers support encapsulations, accommodating servers whose code cannot be modified. Although optimistic computation cannot proceed if the encapsulation communicates with one of these servers, correctness is preserved and the computation proceeds once mandated.

In total, the kernel modifications for encapsulation support consist of an additional 65 bits in each process descriptor and approximately 120 lines of C-language code.

B. Running an Encapsulation

The create, mandate, and abort encapsulation requests are issued by optimistic *make* and serviced by an *encapsulation server* process. The encapsulation server allocates a unique *eid* for each encapsulation, and keeps track of all input and output operations performed by an encapsulation. Servers that support encapsulations inform the encapsulation server of the input and output operations performed by an encapsulation in a server-specific manner. These servers determine that a message comes from an encapsulation by checking the message's *eid* tag.

In our current implementation, both the file server and the terminal server support encapsulations. When an encapsulation opens a file for read, the file's timestamp is recorded with the encapsulation server. When an encapsulation opens a file for write, the request is first recorded with the encapsulation server, then a *hidden file* is created and all subsequent writes are redirected to that file. Hidden files do not appear in the file system directory structure and are only accessible through the encapsulation server. Furthermore, the encapsulation server maintains a hidden file system directory tree for each encapsulation, recording the modifications made by that encapsulation to the directory structure. The hidden directory tree is also used to record the mapping between the names of files modified by the encapsulation and the corresponding hidden files. When an encapsulation writes to the terminal server, the data to be written are recorded with the encapsulation server. Any other operation sent to the terminal server (including a read) is blocked.

In summary, each server records the operations of an encapsulation in a server-specific manner with the encapsulation server. This allows us to take advantage of the semantics or common usage patterns of certain servers. For instance, the file server only records opens with the encapsulation server, and does not need to record individual reads and writes, an important optimization in our environment where typically multiple reads and writes are performed on each open file.

The encapsulation server transfers information between encapsulations during a create encapsulation request if input encapsulations are specified. When a single input encapsulation is specified, the hidden directory tree containing the modifications of the input encapsulation is inherited by the new encapsulation. When more than one input encapsulation is specified, the hidden directory trees of all input encapsulations are merged and then passed on to the new encapsulation.

C. Mandating an Encapsulation

When an encapsulation is mandated, the encapsulation server inquires with the relevant servers (in our implementation, with the file server) whether the timestamps of the inputs that the encapsulation has read have remained unchanged. If so, it instructs the servers to make the outputs performed by the encapsulation visible in the same order as they were recorded. Servers *synchronously* record the output operations of encapsulations with the encapsulation server. Hence, the order in which the outputs are recorded, and thus made visible, is a serialization of the order in which they were created. After all outputs have been made visible, if the encapsulation is still running, the encapsulation server zeroes the *eid* of the encapsulated process and all its descendents, converting the encapsulation into a normal computation. As a result, blocked messages to servers not supporting encapsulations are now delivered.

D. Encapsulation Performance for Make

For the kind of computations that are commonly part of our *makes*, the overhead of executing an encapsulation compared to a normal computation is roughly proportional to the number of file *opens*, as opposed to the number of reads or writes. In our implementation, on SUN-3/50 workstations, the encapsulation overhead is 18 ms per open for read and 8 ms per open for write, for the first open of each file. The encapsulation overhead is lower if the same file is opened again: 10 ms per open for read and 4 ms per open for write. The overhead is lower on subsequent opens because the hidden file system directory tree does not need to be updated. Most 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 directory tree. An implementation in which 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 obtaining a number of timestamps for examination in a single operation. We measured a mandate overhead of 8 ms per open for read and 31 ms per open for write. These times are limited by the time it takes our file server to find a file in the directory tree 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 add to the computation's response time.

E. Implementation Considerations in Other Systems

The V-System facilitated the implementation of encapsulations in two ways. First, we had access to the source, so that we could easily modify the kernel, the file server, and the terminal server. Second, the modular structure of the V-System kept the modifications to the kernel and the servers small, with the bulk of the implementation residing in a separate encapsulation server. We now briefly speculate on how to implement encapsulations in an environment where those two conditions are not fulfilled. To make the discussion specific, we consider an implementation in a Unix environment [16].

If modifications to the operating system are not possible or not desirable, encapsulations could be implemented using a library. The library would define several entry points, such as *read*, *write*, and *open*, which would be called instead of the real kernel calls. Every program that is to be run as an encapsulation would then be linked with this library. The obvious drawback of such a solution is the lack of transparency, requiring the ability to relink existing programs, and resulting in separate executables for normal computations and encapsulations.

If the kernel can be modified, then a transparent Unix implementation appears possible. A bit would need to be added to the process descriptor record indicating the computation runs as an encapsulation. For operations on the file system, the kernel would have to implement the same functionality implemented in the V-System in the encapsulation server, including the hidden file system tree. Blocking an encapsulation on input from, and output to, the terminal would require minor kernel modifications, where many of the existing features of signals could be adapted for this purpose. Buffering the output until mandate time and allowing the encapsulation to continue would also require kernel changes, such as redirecting terminal output to a temporary file until the encapsulation is mandated, then writing the contents of this file to the terminal and allowing subsequent writes to go directly to the terminal. Other terminal operations would likewise either need to be masked or block the encapsulation until mandate time.

IV. Measurements

A. Measurement Environment

The system used for measurement consists of between 8 and 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 run the V-System [4]. Remote execution of programs is transparent and incurs a negligible performance penalty. File access is also transparent, and has equal cost from all diskless machines. The availability of other machines on the network can be determined efficiently using the V group communication mechanism [5].

These 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.

B. 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 time of the *make* request and the latest timestamp of any of the target's sources.
- Command execution time: the running time of each program executed as part of the *make*. All times are normalized to SUN-3 CPU speed.
- The shape of the dependency graph and the number of commands executed as part of the *make*.
- The number of encapsulations aborted as part of each optimistic *make*.

We gathered these *make* statistics for more than 6 months, over which time we measured approximately 4000 requests.

C. Measurement Results

Fig. 2 shows the cumulative distribution of the target out-of-date times. The median and mean values of this distribution are 32 and 378 s, respectively. This implies that the *make* request for most targets is issued fairly soon after a change to the source files is made, but occasionally, users wait much longer before issuing a *make* request. Fig. 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 execution times for each command. We speculate that this is due to the fact that many *make* requests with a small number of commands (and especially those with one command) terminate quickly due to compilation errors.

Most of our *makefiles* have a similar dependency graph (see Fig. 4): a number of independent commands (usually compilations) followed by a single command (usually a linkage). The distribution of the number of commands per *make* is given in Fig. 5. The median number of commands per *make* request is 2, usually 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.

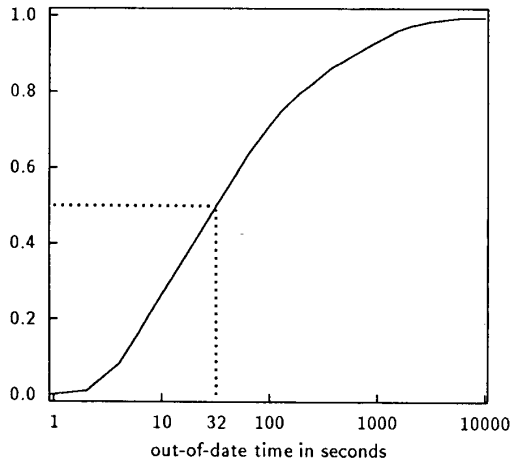


Fig. 2. Cumulative distribution of target out-of-date times.

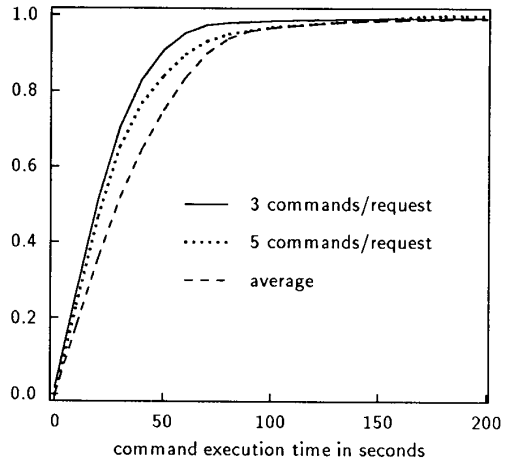


Fig. 3. Cumulative distribution of command execution times.

D. Overhead Estimates

Optimistic *make* uses more system resources (CPU time, device input/output bandwidth, and memory space) than pessimistic *make* due to the presence of aborted optimistic commands and the encapsulation overhead. Table I shows the number of commands mandated and aborted with optimistic *make* in our measurements.

For each mandated command (that is, for every command also necessary in pessimistic *make*), an average of 1.39 optimistic commands are started. Hence, aborted commands impose an extra load of at most 39%. This is an upper limit on the extra load since many of the aborted commands do not run to completion, and thus use fewer resources. For the types of computations considered in this paper (compilations and linkages), a conservative estimate for the encapsulation overhead, derived from measurements, is 2 s per computation during execution and 1 s per computation at mandate time,

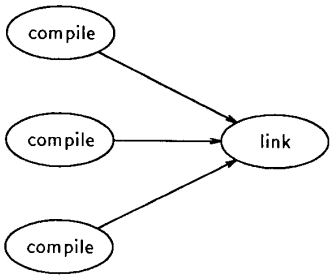


Fig. 4. Typical *makefile* dependency structure.

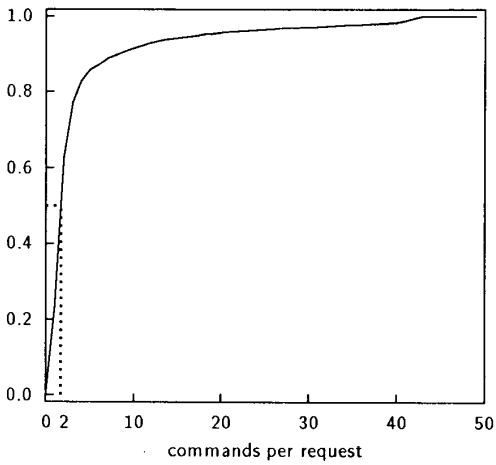


Fig. 5. Cumulative distribution of number of commands per request.

or on average less than 5% of the computation time. Hence, we estimate that the total extra load is at most 44%, and is in practice significantly lower. This extra load is small compared to the large idle times that have been observed in workstation environments, even during peak usage periods [15].

Encapsulations use additional disk space beyond that used by normal commands to store the hidden files. To estimate how much extra space might be used, we assume that each user has a completed, unmandated optimistic *make* request containing the measured average of 4.39 commands. These commands normally consist of a linkage (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 *make* requests requires a total of 81 kilobytes. If we conservatively assume the typical file server has at least 10 megabytes per client, this represents less than 1% of the client's disk allocation.

V. Simulation

To further evaluate the performance of optimistic *make*, we now use the measurements of Section IV to parameterize a simulation of a software development environment. The simulation model consists of N identical machines and M users. Each user issues *make* requests, with the think time between

TABLE I
MANDATED AND ABORTED COMMANDS

Commands	Number	Percent
mandated	16634	100%
aborted	6448	39%
total	23082	139%

requests drawn from an exponential distribution. A command may use any of the N machines, although at any time we allow only a single command to execute on each machine. A centralized scheduler assigns commands to machines in FCFS order, preferring normal commands to optimistic ones. Once a command is started, it runs to completion with no preemption, unless aborted. When all workstations are busy, requests are queued until one becomes available. Simulations with centralized, distributed, preemptive, and nonpreemptive schedulers show that in our environment, under normal load, the choice of scheduling algorithm has little effect [2].

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 Fig. 5, and then select the command execution times from the distribution in Fig. 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 Fig. 4 are considered. For optimistic *make*, we use the same request stream as used for pessimistic *make*, and for each request we draw the out-of-date times for each of the targets from the empirical distribution shown in Fig. 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 commands in optimistic *make*, we introduce an extra command for P percent of the optimistic commands, where P is normally set to the measured value of 39%. We assume both pessimistic and optimistic *make* have negligible request processing overhead. In order to account for encapsulation overhead, each optimistic command is assessed an overhead of 2 s during execution and 1 s at mandate time (see Section IV-D).

The purpose of the simulation is to determine the *response time improvement* of optimistic *make* over pessimistic *make*. *Response time* is the elapsed time from the time when the *make* request is issued to the time when all the commands corresponding to that *make* request are completed. *Response time improvement* is the ratio of response time in pessimistic *make* to response time in optimistic *make*. This is our performance metric: it indicates the relative improvement perceived by the user as a result of optimistic *make*. Since the response time 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 response time improvement of optimistic over pessimistic *make*. 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 95% confidence interval on the median response time improvement for each run with a relative precision of ±3%.[1] This typically requires between 10 and 100 runs of the simulator.

VI. SIMULATION RESULTS

A. The Baseline System

Fig. 6 shows cumulative distributions for the response time improvement in a system similar to our environment. All simulation inputs are drawn from the empirical distributions, the number of machines is set to 10, and the mean think time is set to 6 min. Results are shown for 1, 5, and 10 users. For the 5-user curve, the median response time improvement is 1.72, and the mean improvement is 8.28. The median improvement in the curves for 1 and 10 users is similar, but the mean improvement varies slightly. The shape of the curves reflects the fact that most *make* requests are issued shortly after changes to the source files are made. Improvements are occasionally very large, when all optimistic commands have completed by the time of the *make* request. In this case the response time for the optimistic *make* is equal to the time necessary to mandate the commands. A small percentage of optimistic requests perform worse than the same request in the pessimistic simulation, particularly under high load.

Fig. 7 shows cumulative distributions of response time improvement in the baseline system with varying mean think times. Decreasing the think time affects the improvement more than varying the number of users, in part because shorter think times imply that users do not wait as long before issuing a *make* request after modifying source files, leaving less time to complete the optimistic work. However, the effect of varying the think time on the response time improvement curves becomes minimal for think times larger than 6 min. Measurements indicate that the mean think time in our environment is at least 6 min. Hence, for the remaining experiments described in this paper, we fix its value at 6 min.

Validation: To validate the simulation model, we compare the cumulative response time distribution measured in our implementation to the one obtained from the simulation, for both optimistic and pessimistic *make* (see Fig. 8). We compare response times rather than response time improvements since the improvement, as it is computed in the simulator, cannot be measured from the implementation: each real *make* request is either pessimistic or optimistic, but not both (as in the simulator).

Correlations: The response time improvement is correlated with the number of commands per *make* request and with the total CPU demand per request. Fig. 9 shows the median improvement plotted as a function of the number of commands per *make* request, and Fig. 10 shows the median improvement plotted as a function of the CPU demand per request. With one or two commands per request and with a total CPU demand less than 30 s, the median improvement is noticeably higher

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

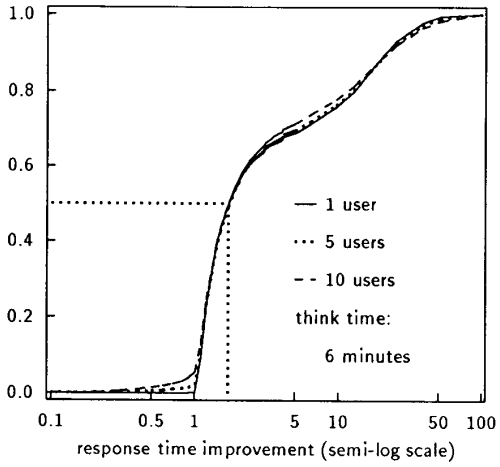


Fig. 6. Cumulative distribution of response time improvement in baseline system (varying users).

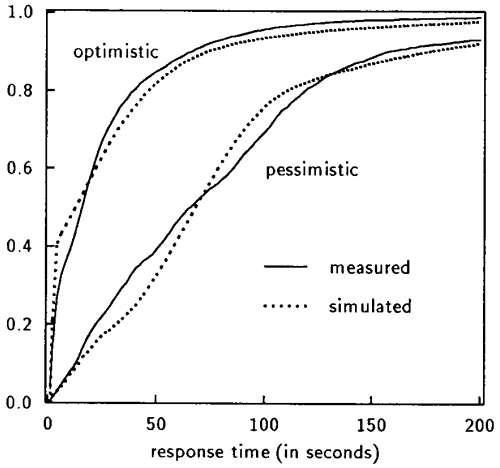


Fig. 8. Cumulative distributions of simulated and measured response times.

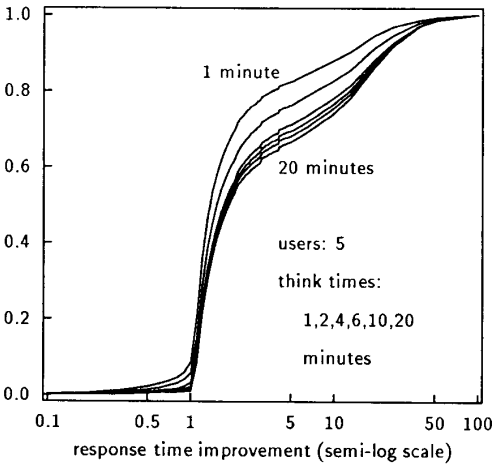


Fig. 7. Cumulative distribution of response time improvement in baseline system (varying think time).

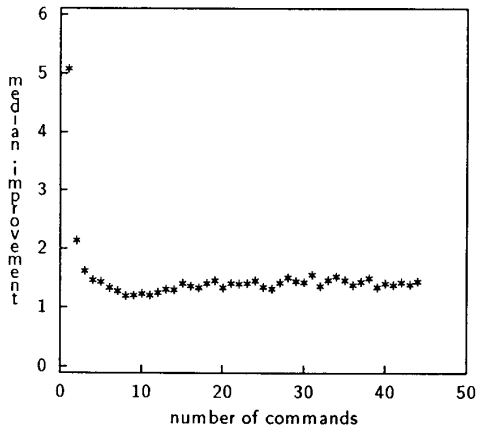


Fig. 9. Median improvement as a function of the number of commands per request.

than with more commands and larger CPU demands. This is because commands are shorter in these cases and thus more likely to have completed optimistically when the *make* request is issued.

Discussion: 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 the computation is mandated. To isolate the effect of changing this ratio from the effect of the number of machines available for execution, we initially assume an infinite number of machines and alternately vary the command execution and out-of-date times (Sections VI-B and VI-C). In Section VI-D, we compare the machine utilization of pessimistic and optimistic *make*, and then address the effect of limiting the number of machines in Section VI-E. Finally, the effect of heterogeneous machine speeds on the response time improvement is studied in Section VI-F.

B. Varying Machine Speeds

To assess the effect of varying machines speeds, the number of machines is set to infinity, and the command execution times (from Fig. 3) are divided by a scale factor. Encapsulation overhead is also reduced by the same factor. Other inputs to the simulation (out-of-date times, think time, and number of commands per *make* request) are as in the baseline model.[2]

Fig. 11 shows the cumulative distribution of response time improvement for the original machine speed (labeled SUN-3), and for systems 8 and 16 times faster (labeled 8*SUN-3 and 16*SUN-3, respectively). Fig. 12 shows the median response times for both pessimistic and optimistic *make* plotted

[2]The number of users is irrelevant with an infinite number of machines.

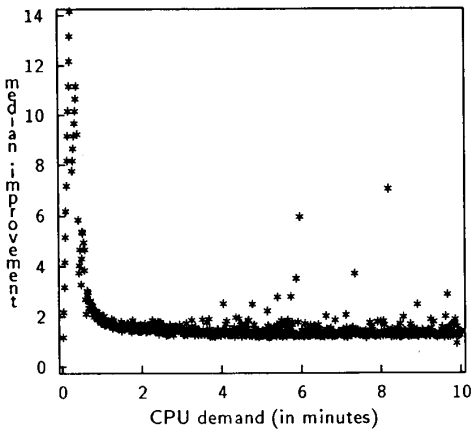


Fig. 10. Median improvement as a function of the CPU demand per request.

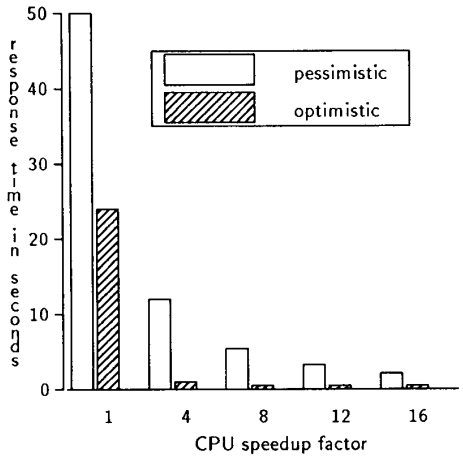


Fig. 12. Median response times for varying machine speeds.

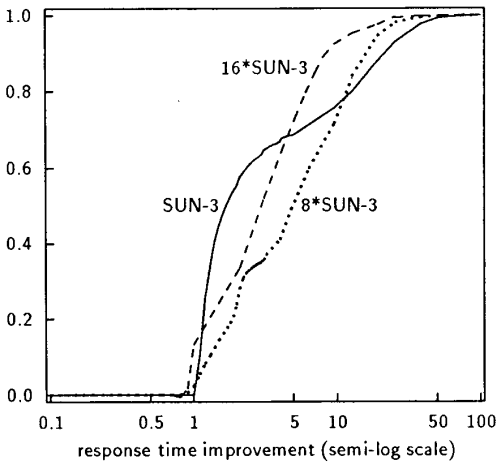


Fig. 11. Cumulative distribution of response time improvement for varying machine speeds.

side-by-side for several CPU speeds.[3] These figures show that as machine speed increases, the difference between the response time of 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 median 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 SUN-3 to 8*SUN-3, many more optimistic commands are completed or are near completion by the time the *make* request is issued. Hence, response time for optimistic *make* is greatly improved. Response time for pessimistic *make* does not improve as fast as for optimistic *make*, yielding a higher response time improvement. Beyond the CPU speed at which most optimistic commands are completed by the time of the *make* request, there is little additional improvement in optimistic *make's* response time. Pessimistic *make* continues to improve, though, decreasing the response time improvement.

[3]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).

C. Varying Out-of-Date Times

While measuring our system, we observed that the median and mean of the out-of-date time distribution changed slightly between different measurement periods. This change corresponds to users waiting longer and shorter time periods, on average, between saving files that necessitate *makes* and typing the *make* command. We simulate this effect by using values drawn from the empirical out-of-date time distribution multiplied by different scale factors. Other simulation inputs are as in the baseline system, with an infinite number of machines.

Fig. 13 shows the cumulative distributions for scale factors of 0.25, 1, and 4. Fig. 14 shows the median response times for both pessimistic and optimistic *make* for several scale factors between 0.25 and 8. Unlike with increasing machine speed (Section VI-B), larger out-of-date times increase both the response time improvement and the difference between median response times, until most optimistic commands are completed by mandate time. With even larger out-of-date times, both remain constant, again in contrast with Section VI-B.

D. Machine Utilization

Fig. 15 shows the probability distribution for the number of busy machines with optimistic *make* using 39% aborted commands (the percentage measured, Section IV-D). This distribution is obtained by sampling the number of busy machines once per minute of simulation time. The number of users is varied between 1 and 16, the mean think time is kept constant at 6 min, other inputs are drawn from the empirical distributions, and an infinite number of machines are available. Simulations with a constant number of users and varying think times give similar results.

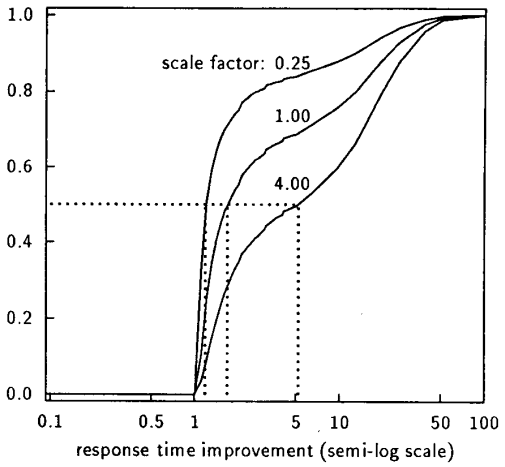


Fig. 13. Cumulative distribution of response time improvement for varying out-of-date scale factors.

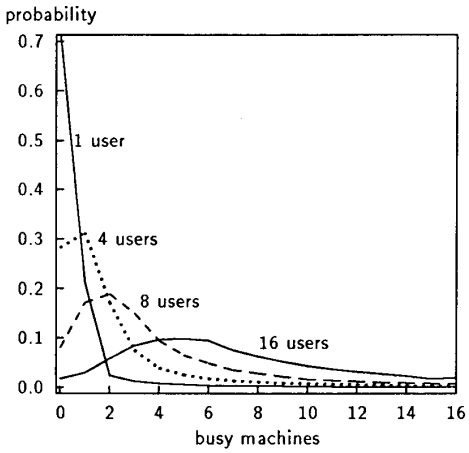


Fig. 15. Probability distribution of busy machines for varying numbers of users.

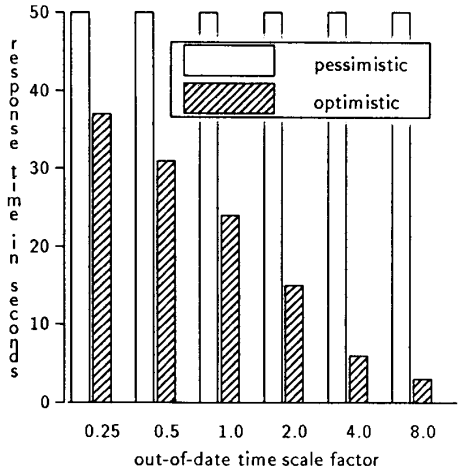


Fig. 14. Median pessimistic and optimistic response times for varying out-of-date scale factors.

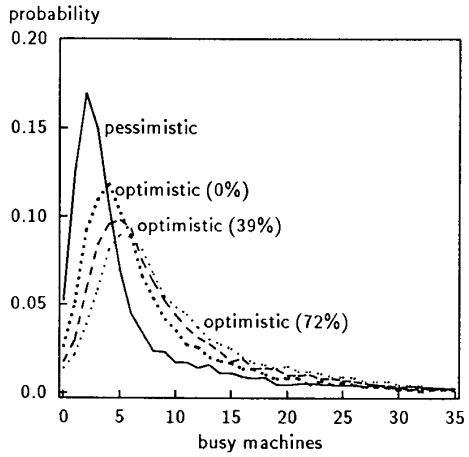


Fig. 16. Probability distribution of busy machines for varying percentages of aborted commands.

Fig. 16 shows the probability distribution of the number of busy machines for 16 users with pessimistic *make*, optimistic *make* with no aborted commands, optimistic *make* with the measured 39% aborted commands, and optimistic *make* with 72% aborted commands (where all source node commands in the *makefile* dependency graph are aborted once). This figure 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 use for each request by using machines as soon as files are modified.[4] The aborted commands add to the overall machine utilization of optimistic *make*, but CPU use remains less variable.

[4]There is a similarity here between our work and load sharing [8]. Load sharing improves throughput by spreading out the workload over different machines. Optimistic execution improves response time by spreading out the workload over time.

E. Limiting the Number of Machines

We now limit the number of machines, while fixing the number of users at 16 and the mean think time at 6 min. All other simulation inputs are taken from the empirical distributions. Fig. 17 shows the response time improvement distribution with 8, 16, and an infinite number of machines. Fig. 18 shows the median response times for optimistic and pessimistic *make* for the same numbers of machines using the three abort ratios from above.

Going from an infinite number of machines to 16, the improvement changes little, since neither optimistic nor pessimistic *make* are machine-limited. When further decreasing the number of machines to 8 (with 2 users per machine), the improvement declines because optimistic commands are

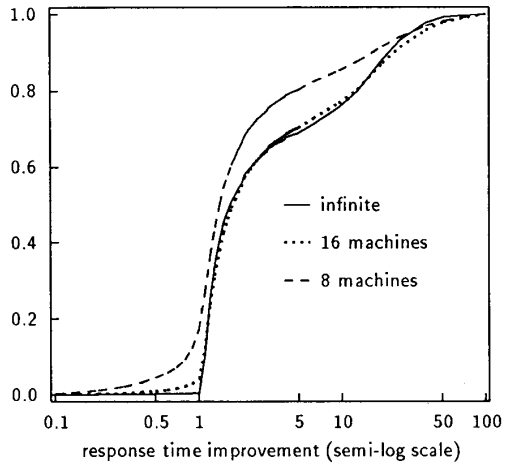


Fig. 17. Cumulative distributions of response time improvement for varying numbers of machines.

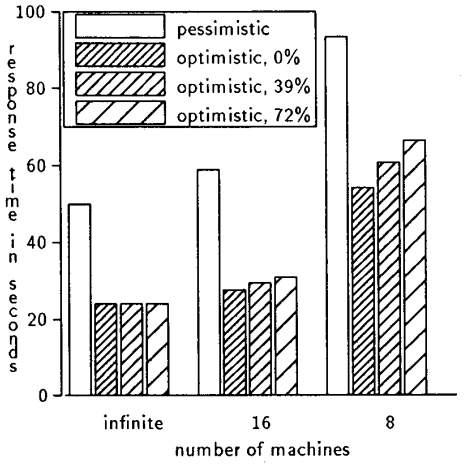


Fig. 18. Median response times for varying numbers of machines.

frequently blocked while normal commands use all the resources. The lack of change 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.

F. Effects of Heterogeneous Machine Speeds

In Section VI-B, we showed that increased machine speed improves the response time improvement of optimistic *make*. We now look at how this improvement is affected in a heterogeneous environment, in which not all machines are of the same speed. We assume two types of machines: slow machines, at the speed used in our baseline system (SUN-3/50), and fast machines, at twice this speed. In a heterogeneous environment, the scheduler prefers fast machines and thus executes commands on fast machines whenever possible. The number of fast machines is varied from 0 to 10 for different numbers of users on a 10-machine system. The remaining parameters are set as in the baseline system with a 6 min mean think time. The resulting median response time improvements are shown in Table II.

TABLE II
MEDIAN RESPONSE TIME IMPROVEMENTS FOR A VARYING NUMBER OF FAST MACHINES IN A 10-MACHINE SYSTEM

	Number of fast machines					
users	0	2	4	6	8	10
1	1.72	2.88	2.81	2.79	2.76	2.73
2	1.71	2.74	2.70	2.66	2.65	2.61
4	1.68	2.62	2.69	2.67	2.66	2.64
6	1.70	2.50	2.65	2.68	2.67	2.66
8	1.71	2.38	2.62	2.69	2.71	2.70
10	1.69	2.24	2.52	2.66	2.70	2.69

With 20% fast machines, the median improvement increases substantially, but with more fast machines, the improvement either levels off or increases only a small amount before leveling off (depending on the number of users). The reason for such a large improvement with a small number of fast machines is twofold. First, with faster machines, optimistic commands are more likely to be completely executed when requested. Second, with optimistic *make*, executions are distributed over time, allowing a larger percentage of commands to be executed on fast machines. For example, when a user types *make* using pessimistic *make*, all commands are started at once (limited by available resources). If the total number of commands is large, several of these are likely to run on slow machines. With optimistic *make*, the user modifies files over time, allowing commands to start at different times. It is likely that a later modification will occur after a previous optimistic command has already completed execution. Thus, a fast machine will be available where a slow one would have been used with pessimistic *make*.

VII. RELATED WORK

Optimistic computations have been incorporated into the Integral C programming environment developed at Tektronix [18]. 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 of their system is given, and there is no evidence that Integral C conceals the output of optimistic computations, which we consider to be essential.

The many parallel and distributed versions of the *make* program [1], [6], [7], [10], [13], [17] are also related to our work. These programs use the dependency graph for controlling parallel or distributed execution, similar to our distributed and optimistic *make*. However, none of these programs uses any form of optimistic computation. We have shown that significant performance improvements are possible by incorporating optimistic computation. Baalbergen [1] and Cmelik [6] point out some possible pitfalls when using parallel or distributed versions of *make* on *makefiles* developed in a sequential environment, and some methods for avoiding them. These pitfalls and the suggested methods for avoiding them also apply to optimistic *make*.

The work on eager evaluation in functional programming languages is, to a lesser extent, related to our work [3], [11], [12]. The functional nature of the language obviates the need for explicit concealment of side effects. Our work differs in that we explicitly deal with outputs, 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 computation increases significantly, since the overhead involved in concealing outputs becomes relatively less important.

VIII. Conclusions and Future Work

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 1.72 and the mean improvement is 8.28. With faster machines, the median improvement grows significantly, until all optimistic commands are completed by the time the user types *make*. The amount of extra resource use resulting from optimistic *make* is small. Given the increased availability of machines and the observed large idle time percentages in many workstation environments, the extra utilization does not adversely affect performance.

We have introduced the notion of encapsulations as the basic construct used in our implementation of optimistic *make*. We are investigating extensions of encapsulations suited to support optimistic computation in general. For instance, optimistic rollback recovery methods [14], [19] appear to require a more incremental notion of encapsulations that allows partial mandates and aborts.

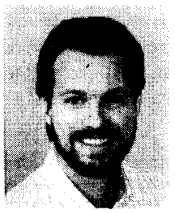
We are also interested in investigating the performance of optimistic *make* with different workloads, for instance workloads measured at other sites or workloads in which program development no longer predominates. Also of interest are the performance implications of different implementations of optimistic *make*, for instance in a system where modification of the kernel and servers is not possible and encapsulation support has to be provided through a library that is linked in with user programs.

Acknowledgment

We would like to thank the referees and the members of the Distributed Systems Group at Rice (J. Carter, E. Elnozahy, J. Fowler, P. Keleher, D. Johnson, and M. Mazina) for their help in improving both the contents and the clarity of the presentation of this paper.

References

[1] E. H. Baalbergen, "Design and implementation of parallel make," *Comput. Syst.*, vol. 1, no. 2, pp. 135–158, Spring 1988.
[2] R. G. Bubenik, "Optimistic computation," Ph.D. dissertation, Rice Univ., May 1990.
[3] F. W. Burton, "Speculative computation, parallelism, and functional programming," *IEEE Trans. Comput.*, vol. C-34, pp. 1190–1193, Dec. 1985.
[4] D. R. Cheriton, "The V distributed system," *Commun. ACM*, vol. 31, no. 3, pp. 314–333, Mar. 1988.
[5] D. R. Cheriton and W. Zwaenepoel, "Distributed process groups in the V kernel," *ACM Trans. Comput. Syst.*, vol. 3, no. 2, pp. 77–107, May 1985.
[6] B. Cmelik, "Concurrent make: The design and implementation of a distributed program in Concurrent C," in *Concurrent C Project*, AT&T Bell Laboratories, Murray Hill, NJ, 1986.
[7] DYNIX, DYNIX Make Manual Page, in *DYNIX Programmer's Manual—Revision 1.15*. Aug. 1987.
[8] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load balancing in homogeneous distributed systems," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 662–675, May 1986.
[9] S. Feldman, "Make—A computer program for maintaining computer programs," *Software Practice and Experience*, vol. 9, no. 4, pp. 255–265, Apr. 1979.
[10] G. S. Fowler, "The fourth generation make," in *Proc. 1985 Summer USENIX Conf.*, June 1985, pp. 159–174.
[11] R. H. Halstead, "Parallel symbolic computing," *IEEE Comput. Mag.*, vol. 19, pp. 35–43, Aug. 1986.
[12] P. Hudak and L. Smith, "Para-functional programming: A paradigm for programming multiprocessor systems," in *Proc. Thirteenth Annu. Symp. Principles of Programming Languages*, Jan. 1986, pp. 243–254.
[13] A. Hume, "Mk: A successor to make," Tech. Rep. 141, AT&T Bell Laboratories, Murray Hill, NJ, Nov. 1987.
[14] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and checkpointing," *J. Algorithms*, vol. 11, no. 3, pp. 462–491, Sept. 1990.
[15] M. W. Mutka and M. Livny, "Scheduling remote processing capacity in a workstation-processor bank network," in *Proc. Seventh Int. Conf. Distributed Comput. Syst.*, Sept. 1987, pp. 2–9.
[16] J. L. Peterson and A. Silberschatz, *Operating System Concepts*, second edition. Reading MA: Addison-Wesley, 1985, ch. 14, pp. 507–533.
[17] E. S. Roberts and J. R. Ellis, "Parmake and Dp: Experience with a distributed, parallel implementation of make," in *Proc. Second Workshop Large-Grained Parallelism*. Available as Special Report CMU/SEI-87-SR-5, Carnegie-Mellon Univ. Software Engineering Institute, Pittsburgh PA, Oct. 1987.
[18] G. Ross, "A practical environment for C programming," in *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. Practical Software Development Environments*, Jan. 1987, pp. 42–48. Also available as *SIGPLAN Notices*, vol. 22, no. 1, Jan. 1987.
[19] R. E. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 3, pp. 204–226, Aug. 1985.



Rick Bubenik (S'87–M'89) received the B.S. degrees in computer science and electrical engineering in 1984, and the M.S. degree in computer science in 1985, all from Washington University, St. Louis, MO. In 1990, he received the Ph.D. degree in computer science from Rice University, Houston, TX.

Since December 1989, he has been a research associate at Washington University. His research interests include protocols for high speed connection-oriented networks and distributed operating systems.

Willy Zwaenepoel (S'81–M'84) received the B.S. degree from the State University of Ghent, Belgium, in 1979, and the M.S. and Ph.D. degrees from Stanford University in 1980 and 1984.

His research interests are in distributed operating systems and in parallel computation. While at Stanford, he worked on the first version of the V kernel, including work on group communication and remote file access performance. At Rice, he has been working on fault tolerance, protocol performance, optimistic computations, and distributed shared memory.

Dr. Zwaenepoel is a member of the Association for Computing Machinery and the IEEE Computer Society.