Using System-Level Models to Evaluate I/O Subsystem Designs

Gregory R. Ganger, *Member*, *IEEE Computer Society*, and Yale N. Patt, *Fellow*, *IEEE*

Abstract—We describe a **system-level simulation model** and show that it enables accurate predictions of both I/O subsystem and overall system performance. In contrast, the conventional approach for evaluating the performance of an I/O subsystem design, which is based on standalone subsystem models, is often unable to accurately predict performance changes because it is too narrow in scope. In particular, conventional methodology treats all I/O requests equally, ignoring differences in how individual requests' response times affect system behavior (including both system performance and the subsequent I/O workload). We introduce the concept of **request criticality** to describe these feedback effects and show that real I/O workloads are not approximated well by either open or closed input models. Because conventional methodology ignores this fact, it often leads to inaccurate performance predictions and can thereby lead to incorrect conclusions and poor design choices. We illustrate these problems with real examples and show that a system-level model, which includes both the I/O subsystem and other important system components (e.g., CPUs and system software), properly captures the feedback and subsequent performance effects.

Index Terms—I/O subsystems, storage subsystem, system-level model, system simulation, disk system, disk scheduling, simulation, performance model, disk modeling.

1 Introduction

In response to the growing importance of I/O subsystem performance, researchers and developers are focusing more attention on the identification of high-performance I/O subsystem architectures and implementations. The conventional approach to evaluating the performance of a subsystem design is based on standalone subsystem models (simulation or analytic). With this approach, a model of the proposed design is exercised with a series of I/O requests. The model predicts how well the given design will handle the given series of requests using subsystem performance metrics.

The ability of any performance evaluation methodology to identify good design points depends upon at least three factors:

1) the accuracy of the model,
2) the representativeness of the workload, and
3) how well the performance metrics translate into overall system performance.

The first two factors relate to the accuracy of the performance predictions and the third relates to their usefulness. I/O subsystem model accuracy can be achieved by careful calibration against one or more real subsystems. The latter two factors, however, represent important flaws in conventional methodology:

1) The workloads used are often not representative of reality in that they do not accurately reflect feedback effects between I/O subsystem performance (in particular, individual request completion times) and the workload (in particular, subsequent request arrivals).
2) Changes in I/O subsystem performance (as measured by response times and throughput of I/O requests) do not always translate into similar changes in overall system performance (as measured by elapsed times or throughput of user tasks).

These problems are fundamental to a subsystem-oriented approach and are independent of the model's accuracy.

Both problems arise because conventional methodology tends to treat all I/O requests as equally important. In reality, however, different requests affect overall system performance in different ways and to different degrees. Some performance effects, such as increased system bus and memory bank contention, depend mainly on the quantity and timing of accesses. Others, such as false idle time, are highly dependent on whether and when processes wait for I/O requests. **False idle time** is that time during which a processor executes the idle loop because all active processes are blocked waiting for I/O requests to complete. This is different from regular idle time, which is due to a lack of available work in the system. False idle time is a significant concern, as it completely wastes the CPU for some period of time (independent of processor speed) rather than some number of cycles.

We describe three distinct classes of **request criticality** based on how individual requests' response times (i.e., the times from issue to completion) affect I/O wait times [1]. Generally speaking, one request is more critical than another if it is more likely to block application processes and thereby waste CPU cycles. Most real workloads consist of a

- *G.R. Ganger is with the Departments of Electrical and Computer Engineering and Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890. E-mail: ganger@ece.cmu.edu.*
- *Y.N. Patt is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122. E-mail: patt@eecs.umich.edu.*

Manuscript received December 1996; revised March 1998.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 102372.

mixture of requests from the three classes. The common approaches to I/O workload generation fail to accurately recreate the effects of request criticality mixtures. Also, most I/O performance metrics do not reflect variations in request criticality.

To directly address these problems, we propose a new methodology based on system-level models [1]. A **system-level model** includes I/O subsystem components as well as enough other system components to

1) accurately incorporate feedback effects between I/O performance and system behavior and
2) directly provide overall system performance metrics.

Simulation times can remain reasonable because the granularity of system component events can correspond to that of I/O subsystem events (i.e., hundreds of μs to tens of ms). A well-designed system-level model will eliminate the two problems outlined above.

The remainder of the paper is organized as follows. Section 2 introduces the concept of request criticality. Section 3 describes system-level modeling. Section 4 describes previous work related to request criticality and system-level models. Section 5 describes and validates our simulation infrastructure. While many of the concepts in this paper apply to other forms of I/O, such as networks and user interfaces, and other levels of the memory hierarchy, such as processor caches and tertiary storage, our simulator and experiments focus on the secondary storage subsystem. Section 6 illustrates the first problem outlined above with concrete examples where conventional methodology leads to incorrect conclusions (quantitative and qualitative) regarding storage subsystem performance by failing to properly incorporate feedback between request completions and subsequent arrivals. Section 7 briefly investigates criticality-based disk scheduling. The results illustrate the second problem outlined above; system performance increases while storage subsystem performance decreases. Section 8 draws conclusions and suggests avenues for future research.

2 Request Criticality

I/O requests separate into three classes: time-critical, time-limited, and time-noncritical. A request is **time-critical** if the process that generates it must stop executing until it completes. False idle time accumulates if there are no other processes that can be executed when the current process blocks. Examples of time-critical requests include demand page faults, synchronous file system writes, and database block reads.

Time-limited requests are those that become time-critical if not completed within some amount of time (the **time limit**). File system prefetches are examples of this. Time-limited requests are similar to time-critical requests in their effect on performance. The major difference is that they are characterized by a time window in which they must complete in order to avoid I/O wait times. If completed within this window, they cause no process to block. Time-limited requests are often speculative in nature (e.g., prefetches). When prefetched blocks are unnecessary, performance degradation (e.g., resource contention and cache pollution) can result.

No process waits for **time-noncritical** requests. They must be completed to maintain the accuracy of nonvolatile storage, to free resources (e.g., memory), and/or to allow background activity to progress. Examples of time-noncritical requests are background flushes and requests issued for background data reorganization. Except when main memory saturates, time-noncritical requests impact performance indirectly. For example, they can interfere with the completion of more critical requests, causing additional false idle time. Also, delays in completing time-noncritical requests can reduce the effectiveness of the in-memory disk cache.

In practice, there are no truly time-noncritical requests. If background activity is never completed, some application process will eventually block and wait. For example, if background disk writes are not handled, main memory will eventually consist entirely of dirty pages and processes will have to wait for some writes to complete. However, the time limits are effectively infinite in most real environments because of their orders of magnitude (e.g., many seconds in a system where a request can be serviced in tens of milliseconds). Given this, it is useful to make a distinction between time-limited requests (characterized by relatively small time limits) and time-noncritical requests (characterized by relatively long time limits).

Time-noncritical requests can have completion time requirements—for example, if the guarantees offered by the system require that new data reach stable storage within a specified amount of time. Such requests are not time-limited according to our definition,[1] but the I/O subsystem must be designed to uphold such guarantees. Fortunately, these time constraints are usually sufficiently large to present no problem.

Most real workloads consist of a mixture of time-critical, time-limited, and time-noncritical requests. For example, extensive measurements of three different UNIX systems ([2]) showed that time-critical requests range from 51-74 percent of the total workload, time-limited requests range from 4-8 percent, and time-noncritical requests range from 19-43 percent. For the workloads used in this paper, time-critical requests comprise 1-95 percent of the total workload, time-limited requests comprise 0-70 percent, and time-noncritical requests comprise 5-74 percent.

3 System-Level Models

Rather than focusing on the storage subsystem in a vacuum, a **system-level model** focuses on the system as a whole and the interactions between components, so as to incorporate the effects of request criticality mixtures. Very simply, a system-level model consists of modules for each major system component and its interfaces to the outside world (e.g., users and other systems).[2] Processes execute within a system-level model in a manner that imitates the behavior of the corresponding system. Also, external

1. This is one of several system-behavior-related I/O workload characteristics that are orthogonal to request criticality. Each such characteristic represents another dimension in a full I/O request taxonomy.

2. Users and other systems could both be viewed as components of a system-level model, depending upon how comprehensive the model is intended to be.

TABLE 1
EXAMPLE EVENTS IN A SYSTEM-LEVEL MODEL

Scope	Type	Description
External Only	System Call	call to OS for service
	System Return	return from System Call
	Fork	construct/initiate a new process
	Exit	last event of a process sequence
External or Internal	Read Block	access the contents of a specific block
	Write Block	modify the contents of a specific block
	Trap	an exception occurring during execution
	Trap Complete	last event in exception service routine
	Interrupt	an interrupt arrival
Internal Only	Intr Complete	last event in interrupt service routine
	Disk Request	generate a request for disk I/O
	Disk Access	initiate disk I/O
	Context Switch	change the process executed by a CPU
	Wait for Event	temporarily block process execution
	Wake Up	reenable process(es) waiting for event

This table lists a set of representative events for a high-level system-level model, partitioned into three groups based on whether they are externally or internally generated. External events, the causes of system activity, are part of the input workload (entries in process event sequences, except for interrupt arrivals). Internal events are generated by the model in response to particular circumstances and system states. Some events can be either, meaning that they are sometimes explicit in the input workload and sometimes implicitly generated by the model.

interrupts may arrive at the interfaces, triggering additional work for the system.

For the host system part of a system-level model, we have found that high-level system software events are sufficient to capture the first and second order performance effects of storage subsystem designs because of the large granularity of storage subsystem events (e.g., 100s of μs to 10s of ms). Each event represents an important system software state change, possibly affecting the flow of work in the system. Processes and interrupt service routines can be modeled as sequences of events separated by computation times. The CPUs "execute" this software by decrementing each computation time (as simulated time progresses) until it reaches zero, at which time the next event "occurs" (i.e., the system state is changed). If an interrupt arrives before the computation time reaches zero, then the computation time is updated, the new interrupt is added to the (possibly empty) stack, and the first event of the service routine becomes the CPU's current event. Interrupt completion events remove interrupts from the stack and context switch events replace the current process. I/O request events initiate activity in the storage subsystem components of the simulator.

The key aspect of a successful system-level model is the distinction between causes of system activity and effects (i.e., the resulting system activity). For example, a system call to read file data is a cause. Corresponding storage activity, context switches, and completion interrupts are all effects. To correctly incorporate feedback effects (e.g., request criticality), causes must be part of the input workload and effects must not. Also, interactions between processes (e.g., CPU contention) must be part of the model to obtain accurate system performance predictions. Table 1 lists some example events in our simulator. ([3] gives the full list.)

4 PREVIOUS WORK

4.1 Request Criticality

Although we have found no previous work that specifically attempts to classify I/O requests based on how they affect system performance, previous researchers have noted differences between various I/O requests. Many have recognized that synchronous (i.e., time-critical) file system writes generally cause more performance problems than nonsynchronous (i.e., time-limited and time-noncritical) writes [4], [2].

Researchers have noted that bursts of delayed (i.e., time-noncritical) writes caused by periodic update policies can seriously degrade performance by interfering with read requests (which tend to be more critical) [5]. Carson and Setia argued that disk cache performance should be measured in terms of its effect on read requests. While not describing or distinguishing between classes of I/O requests, they did make a solid distinction between read and write requests based on process interaction. This distinction is not new. The original UNIX system (System 7) used a disk request scheduler that gives non-preemptive priority to read requests for exactly this reason. The problem with this approach (and this distinction) is that many write requests are time-limited or time-critical. Such requests are improperly penalized by this approach.

With nonvolatile cache memory, it is easy for storage subsystem designers to translate write latency problems into write throughput problems, which are much easier to handle. This leads directly to the conclusion that read latencies are the most significant performance problem. Researchers are currently exploring approaches to predicting and using information about future access patterns to guide aggressive prefetch activity [6], [7], [8], hoping to utilize high-throughput storage systems to reduce application-observed read latencies. Another way to view these efforts is that they are attempting to increase the time limits associated with prefetches.

Priority-based disk scheduling has been examined and shown to improve overall performance in real-time systems [9], [10] and environments where some tasks are more important than others [11]. In these cases, each request's priority reflects the priority or deadline of the associated process. Criticality-based disk scheduling (see Section 7) should complement priority-based scheduling.

Finally, the *Head-Of-Queue* [12] or *express* [13] request types present in many I/O architectures show recognition of the possible value of giving priority to certain requests. While present in many systems, such support is generally not exploited by system software. Currently, researchers are exploring how such support can be utilized by a distributed disk request scheduler that concerns itself with both mechanical latencies and system priorities [14].

4.2 System-Level Modeling

References [15] and [16] describe system-level modeling efforts used mainly for examining alternative system configurations (as opposed to I/O subsystem designs). Reference [17] describes a system performance measurement technique that consists of tracing major system events. The end purpose for this technique is to measure system performance under various workloads, rather than as input to a simulator to study I/O subsystem design options. However, Haigh's tracing mechanism is very similar to our trace acquisition tool. Reference [18] describes a set of tools for studying I/O performance as part of the entire system. These tools are based on instruction-level traces. While simulating every detail of systems' activity is certainly more accurate than abstracting part of it away, it is not practical given the timing granularity of I/O events. The enormous simulation times and instruction trace storage requirements make this approach both time-prohibitive and cost-prohibitive.

A more recent alternative is full machine simulation, as provided by the SimOS environment [19]. Although this approach does allow the actual operating system and full applications to be run within the simulation environment, it also runs an order of magnitude slower than the raw hardware. More abstract simulators (which are sufficient for storage subsystem evaluations) generally run much faster than the real hardware and can, therefore, produce results more than two orders of magnitude more efficiently than full machine simulation.

There have been a few instances of very simple system-level models being used to examine the value of caching disk blocks in main memory [20], [21], [22]. We expand on these efforts in two ways:

1) by using detailed, validated system-level models, and
2) by using system-level models to evaluate storage subsystem designs as well as host system cache designs.

Recently proposed techniques for replaying traces of file system requests attempt to realistically recreate performance-workload feedback effects by incorporating the observed inter-request user/process think times [23], [24]. Unfortunately, this approach to trace replay is unlikely to be successful with storage I/O request traces, because the host-level cache and background system daemons make it extremely difficult to identify who is responsible for what by simply observing the I/O requests. However, these techniques do offer a healthy supply of input workloads for system-level models, which would, of course, need a module that simulates file system functionality [23]. Thekkath et al. propose such an approach for using simulation to evaluate file system designs.

5 The Simulation Infrastructure

To validate our claims (as well as enable a variety of storage subsystem architecture studies), we constructed a system-level simulator. This section briefly describes the simulator, the system that it is usually configured to emulate, the traces used in the experiments, and results from comparing the simulator's predictions to the corresponding real system. Reference [3] describes all of these in greater detail.

5.1 The Simulator

The simulator can be used as either a standalone storage subsystem model (driven by I/O request traces) or a system-level model (driven by the system-level traces described below). It contains modules for most secondary storage components of interest, including device drivers, buses, controllers, adapters, and disk drives. The storage components are very detailed and can be configured in a wide variety of ways. It also contains the host system component modules necessary to function as a system-level model at a level of detail appropriate for evaluating storage subsystem designs. For system-level model experiments, the storage components are driven by I/O requests generated by the host system components. The key point, with respect to this work, is that the same storage subsystem model can be driven by either simple I/O request traces or activity generated by the system-level components, allowing direct comparisons between the two.

5.2 The Experimental System

The base system for our experiments (both simulation and implementation) is an NCR 3433, a 33 MHz Intel 80486 machine equipped with 48 MB of main memory. For all of the experiments, including trace collection, the available physical memory is partitioned into 40 MB for normal system use and 8 MB for a trace buffer (see below). The disk, an HP C2247, is a high performance, 3.5-inch, 1 GB SCSI storage device [25]. Table 2 lists some basic characteristics of this disk drive and [26] provides a thorough breakdown of simulator configuration parameter values. The operating system is UNIX SVR4 MP, NCR's production operating system for symmetric multiprocessing. The default file system, *ufs*, is based on the Berkeley fast file system [27]. File system caching is well integrated with the virtual memory system, which is similar to that of SunOS [28], [29]. Unless otherwise noted, the scheduling algorithm used by the device driver is LBN-based C-LOOK. All experiments are run with the network disconnected.

One important aspect of the file system's performance (and reliability) is the syncer daemon. This background process awakens periodically and writes out dirty buffer cache blocks. The syncer daemon in UNIX SVR4 MP operates differently from the conventional "30 second sync." It awakens once each second and sweeps through a fraction

TABLE 2
BASIC CHARACTERISTICS OF THE HP C2247 DISK DRIVE

HP C2247 Disk Drive	
Formatted Capacity	1.05 GB
Rotation Speed	5,400 RPM
Data Surfaces	13
Cylinders	2,051
512-Byte Sectors	2,054,864
Zones	8
Sectors/Track	56-96
Interface	SCSI-2
256 KB Cache, 2 Segments	
Track Sparing/Reallocation	

of the buffer cache, initiating an asynchronous write for each dirty block encountered. This algorithm represents a significant reduction in the write burstiness associated with the conventional approach (as studied in [5]) but does not completely eliminate the phenomenon.

5.3 System-Level Traces

5.3.1 Trace Acquisition

To collect traces of system activity, we instrumented the operating system. The instrumentation collects trace data in a dedicated kernel memory buffer and alters performance by less than 0.1 percent, assuming that the trace buffer is not otherwise available for system use. When trace collection is complete, user-level tools are used to copy the buffer contents to a file. Each trace record contains an event type, the CPU number, a high-resolution timestamp (approximately 840 nanoseconds) and optional event-specific information.

A postprocessing tool translates the system event traces into system-level traces that can be used with the simulator. To enable this, and also to obtain configuration information, the instrumentation captures several auxiliary system events (e.g., context switches and storage I/O interrupts). Also, when tracing begins, the initial process control state is copied into the trace buffer. The process or interrupt handler to which each traced event belongs can be computed from this initial state and the traced context switches, interrupt arrivals/completions and CPU numbers. The postprocessing tool passes once through the trace and produces files containing:

1) a set of per-process sequences of event/computation time pairs,
2) a list of external interrupts, including arrival times and sequences of event/computation time pairs for the handlers,
3) the state of the initially active processes,
4) a list the observed disk requests (i.e., a disk request trace),
5) performance statistics for the traced period of activity.

5.3.2 Workloads

While our hypotheses have been tested with a number of workloads (see [3]), we summarize the results in this paper with four workloads that consist of individual, though substantial, user tasks. Tracing for each experiment begins a few seconds before the task of interest is started and continues until the task completes and all I/O requests for file cache blocks dirtied during task execution are initiated by the syncer daemon and completed. We refer to the process that performs the main work as the **task-executing process**. Several instances of each workload have been traced, allowing generation of statistically significant results.

Table 3 lists basic characteristics of the four workloads used in this paper. *compress* uses the UNIX *compress* utility to reduce a 31 MB file to 11 MB. *uncompress* uses the UNIX *uncompress* utility to return an 11 MB file to its original 31 MB size. *copytree* uses the UNIX *cp -r* command to copy a user directory tree containing 535 files totaling 14 MB of storage. *removetree* uses the UNIX *rm -r* command to remove the new copy resulting from an immediately prior *copytree* execution. With the exception of *removetree*, source data are not present in the main memory file block cache when a task begins.

We selected this set of four workloads for two reasons. First, they are all real applications, removing issues related to synthesis. Second, and more importantly, they cover a broad range of request criticality mixes, allowing us to expose a number of problems with conventional methodology. For the workloads in this set, time-critical requests comprise 1-95 percent of the total workload, time-limited requests comprise 0-70 percent, and time-noncritical requests comprise 5-74 percent. Although this set of workloads is not generally representative of system workloads across all characteristics, it covers request criticality mixes well. Further, the results reported and conclusions drawn in this paper are consistent with our experiences and experiments with a number of other workloads, including video servers, WWW servers, and time-sharing systems [3].

5.4 Validation

To verify that the simulator correctly emulates the experimental system, we collected several performance measurements in the form of system-level traces. The simulator was configured to match the experimental system and driven with these traces. The simulator results and the system measurements were then compared using various performance metrics. Table 4 shows the results for the *compress* workload. Most of the simulator values are within 1 percent of the corresponding measured value.

TABLE 3
BASIC CHARACTERISTICS OF THE INDEPENDENT TASK WORKLOADS

Independent Task	Task Elapsed Time	Task CPU Time	I/O Wait Time	# of I/O Requests	Avg. I/O Resp. Time
compress	198 sec.	162 sec.	25.6 sec.	10,844	53.3 ms
uncompress	144 sec.	91.0 sec.	47.1 sec.	10,983	1076 ms
copytree	89.5 sec.	17.6 sec.	69.7 sec	8,995	147 ms
removetree	18.2 sec.	3.05 sec.	14.9 sec	1,176	15.6 ms

TABLE 4
SIMULATOR VALIDATION FOR THE *compress* WORKLOAD

Metric	Measured	Simulated	% Diff.
Elapsed Time	198 sec	195 sec	–1.5%
CPU utilization	81.3 %	82.0 %	0.9%
Number of interrupts	61225	60092	1.9%
Number of I/O requests	10844	10844	0.0%
Number of I/O waits	370	367	–0.8%
Average I/O wait time	69.2 ms	67.7 ms	–2.2%
Average I/O access time	6.43 ms	6.34 ms	–1.4%
Average I/O response time	53.3 ms	54.2 ms	1.7%

TABLE 5
MEASURED AND SIMULATED PERFORMANCE IMPROVEMENT FOR THE *uncompress* WORKLOAD

Metric	Measured Improvement	Simulated Improvement
Elapsed Time	10.4 %	10.6 %
Average I/O response time	65.2 %	66.1 %
Average I/O access time	17.8 %	17.8 %

The improvements come from using C-LOOK disk scheduling rather than First-Come-First-Served.

The largest difference observed among all of our validation experiments was 5 percent. In addition, the validity of our storage subsystem modules has been independently established [26], [30].

Once a simulator's ability to emulate a real system has been verified, an additional level of validation can be achieved by modifying both and comparing the resulting changes in performance. That is, one can measure the change in performance on the real system and compare it to the change in performance predicted with the simulator. Table 5 compares measured and simulated performance improvements resulting from the use of a C-LOOK disk scheduling algorithm, rather than a simple First-Come-First-Served algorithm. The simulator's predictions match the measured values very closely.

6 PERFORMANCE/WORKLOAD FEEDBACK

Conventional methodology fails to properly model feedback effects between request completions and request arrivals. Because almost any change to the storage subsystem or to the system itself will alter individual request response times, the change will also alter (in a real system) the workload observed by the storage system. Because the purpose of most performance evaluation is to determine what happens when the components of interest are changed, lack of proper feedback can ruin the representativeness of a workload, leading to incorrect conclusions regarding performance. This section demonstrates this problem via several examples where inaccurate feedback effects cause the conventional methodology to produce erroneous results.

6.1 Storage Subsystem Workload Generators

Commonly used workload generators for storage subsystem models fall into two groups: open and closed. An **open subsystem model** assumes that there is no feedback between individual request completions and subsequent request arrivals. This assumption ignores real systems' tendency to regulate (indirectly) the storage workload based on storage responsiveness. As a result, workload intensity does not increase (decrease) in response to improvement (degradation) in storage performance. This fact often leads to overestimation of performance changes by reducing (enlarging) queue times excessively. The lack of feedback also allows requests to be outstanding concurrently that would never in reality be outstanding at the same time.

A **closed subsystem model** assumes unqualified feedback between storage subsystem performance and the workload. Request arrival times depend entirely on the completion times of previous requests. The main problem with closed subsystem models is their steady flow of requests, which assumes away arrival stream burstiness (i.e., interspersed periods of intense activity and no activity). As a result, closed subsystem models generally underestimate performance changes, which often consist largely of queuing delays. Also, the lack of both intense bursts of activity and idle periods can prevent the identification of optimizations related to each.

To exercise standalone subsystem models, we use traces of I/O requests extracted from system-level traces. For open subsystem models, the traced arrival times and physical access characteristics are maintained to recreate the observed storage I/O workload. If the storage subsystem model exactly replicates the observed behavior, the resulting storage performance metrics will be identical, as the feedback effects (or lack thereof) will not come into play. Because the results match reality in at least one instance, we have used open subsystem simulation driven by traces of observed disk activity in our previous work. There is no corresponding trace-based workload generator for closed subsystem models, which never match the reality of most workloads.

For closed subsystem models, the traced physical access characteristics are maintained, but the arrival times are discarded in favor of the closed workload model. Each simulation begins by reading N requests from the trace, where N

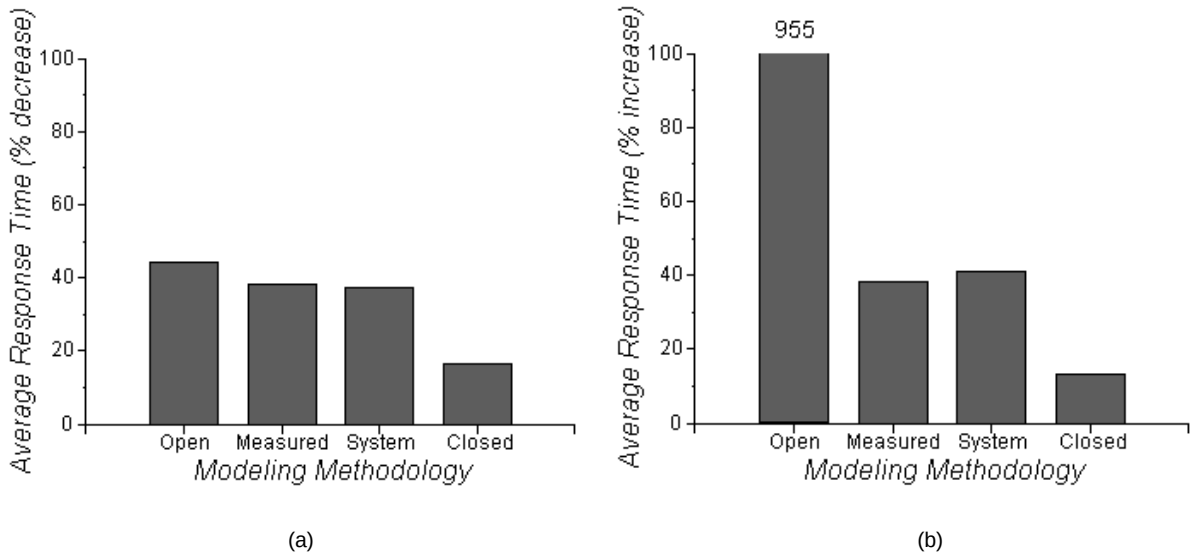


Fig. 1. Scheduling algorithm comparison for the *copytree* workload. The closed subsystem model maintains a request population of 9 for (a) and 5 for (b), corresponding to the average populations observed with FCFS and C-LOOK. In graph (b), the open subsystem model predicts a 955 percent increase in the average response time. (a) FCFS ⇒ C-LOOK. (b) C-LOOK ⇒ FCFS.

is the constant request population, and issuing them into the storage subsystem simulator. As each request completes, a new request is read and issued. The value of N is chosen to match (as closely as possible) the average number of outstanding requests over the duration of the trace. However, the minimum value for N is one because no inter-request think times are being utilized.

6.2 Quantitative Errors

By misrepresenting performance/workload feedback effects, standalone subsystem models frequently overestimate or underestimate performance changes. We illustrate this behavior by comparing two disk scheduling algorithms. The **First-Come-First-Served** (FCFS) algorithm services requests in arrival order. The **C-LOOK** algorithm always services the closest request that is logically forward (i.e., has a higher starting block number) of the most recently serviced request. If all pending requests are logically behind the most recent one, then the request with the lowest starting block number is serviced.

Fig. 1 shows the measured and predicted performance effects of replacing one algorithm with the other for the *copytree* workload. Each figure contains two graphs, representing the change from FCFS to C-LOOK and the reverse. The distinction relates to which algorithm was used on the real system during trace collection. Each graph displays the performance change predicted by an open subsystem model, a system-level model, and a closed subsystem model. Also shown is the measured performance difference, which matches the system-level model's predictions closely in every case (further evidence of the system-level model's accuracy).

As expected, the open subsystem model overestimates the actual performance change and the closed subsystem model underestimates it. This behavior is consistent across all of our modeling methodology comparisons. Note that the quantitative error associated with open subsystem modeling is much larger when the modeled subsystem services requests less quickly than the traced subsystem.

In general, we have found that prediction errors are larger when the workload is heavier and when the real workload contains larger or smaller amounts of request criticality relative to the workload model. The former effect relates to the fact that queue times dominate service times for heavy workloads, and performance/workload feedback most directly affects queue times. The latter effect is more straightforward. An open subsystem model, which assumes that all requests are time-noncritical, will have difficulty recreating a workload that consists mostly of time-critical requests. Similarly, closed subsystem models have difficulty with time-noncritical requests.

6.3 Qualitative Errors

While quantitative errors are discomforting, one might be willing to accept them if the qualitative answers (e.g., design A is superior to design B) were consistently correct. Unfortunately, this is not the case. It is quite easy to construct situations where open and closed subsystem models lead to incorrect qualitative conclusions. For example, an open subsystem model might lead one to believe that collapsing pending requests that overlap can result in large performance increases. In reality, most systems are structured such that overlapping requests are generated one at a time. As another example, the lack of burstiness in closed subsystem models might erroneously lead one to believe that exploiting idle disk time offers no benefits. While these examples may be obvious to many designers, others are not. Trivializing feedback effects, as I/O subsystem workload generators do, can result in incorrect answers because of complex interactions and secondary performance effects.

To illustrate this problem, we evaluate the use of disk cache awareness in aggressive disk scheduling algorithms. The **Shortest-Positioning-Time-First** (SPTF) algorithm uses full knowledge of request processing overheads, logical-to-

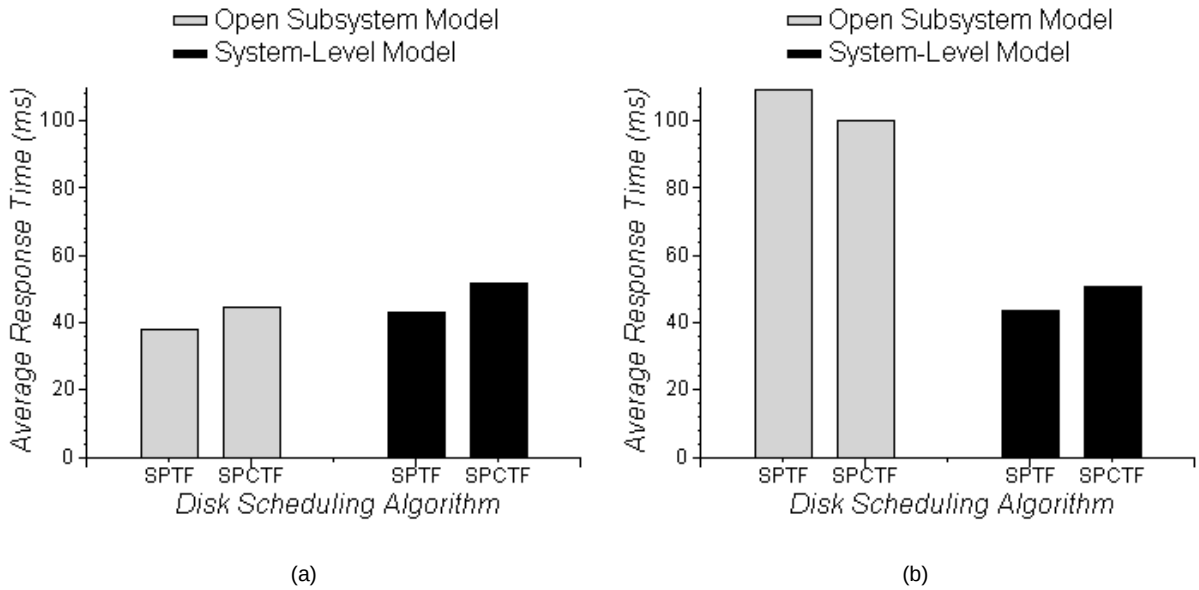


Fig. 2. Cache-Aware Disk Scheduling for the *compress* Workload. (a) Scale Factor = 1.0. (b) Scale Factor = 2.0.

physical data block mappings, mechanical positioning delays, and the current read / write head location to select for service the pending request that will require the shortest positioning time [31], [32]. The SPTF algorithm can be modified to track the contents of the disk's on-board cache and estimate a positioning time of zero for any request that can be serviced from the cache, resulting in the **Shortest-Positioning-(w/Cache)-Time-First** (SPCTF) algorithm [26]. Because our HP C2247's cache performs all writes synchronously, the value of cache-awareness should depend upon the frequency with which read requests that would hit in the cache are otherwise delayed.

Giving preference to requests that hit in the on-board disk cache should improve SPTF performance in several ways. First, elementary queuing theory tells us that servicing the quickest requests first reduces the average queue time, and cache hits can certainly be serviced in less time than requests that access the media. Second, SPTF will often service a set of pending sequential read requests in nonascending order. The SPTF algorithm always selects the request that will incur the smallest positioning delay, so the first request selected from the set may not be the one with the lowest starting address. Further, the second request serviced is often not the next sequential request. During the bus transfer and completion processing of the first request, the media platter rotates past the beginning of the next sequential request (prefetching it, of course). The SPTF algorithm, which is ignorant of the prefetching, may not select the next sequential request. On the other hand, the SPCTF algorithm selects the next sequential request to exploit the prefetch behavior. Third, SPCTF can improve the hit rate when cache segments would otherwise be reused before overlapping requests utilize their contents.

Fig. 2 compares SPTF and SPCTF with the *compress* workload. The two graphs show average response times with scale factors of 1 and 2, respectively. The open subsystem model scales a workload by shrinking the measured request inter-arrival times. The system-level model scales a workload by increasing the speed of the host system components (e.g., increasing the rate at which a CPU decrements computation times). The request arrival rates for the two models are similar (33 and 34 requests per second with a scaling factor of 1 and 67 and 70 requests per second with a scaling factor of 2). Note that the quantitative error in the open subsystem model's predictions increases sharply with workload scaling. Each graph shows two pairs of bars, comparing the average response times predicted with the open subsystem model and the system-level model.

The open subsystem model predicts that cache-awareness increases the average response time by 17 percent for a trace scaling factor of one. This unexpected result is caused by interactions between complex performance / workload feedback effects, which remain largely intact in this case, and the disk's prefetching behavior; this is explained further below. For a trace scaling factor of two, the open subsystem model predicts that the average response time decreases by 8 percent. Most of the performance improvement comes from servicing pending sequential reads in ascending order, thereby exploiting the disk's prefetch behavior. A storage subsystem designer might observe these results and conclude that cache-awareness improves storage subsystem performance under heavy workloads, but can hurt performance under lighter workloads. A hybrid algorithm that uses knowledge of cache contents only when the workload is heavy might then be devised.

Although not shown, the closed subsystem model also predicts that SPCTF outperforms SPTF (by 0.6 percent for request populations of 4, 8, 16, and 32). As with the open subsystem model, the improvement comes mainly from servicing pending sequential reads in ascending order. While the response time reduction is not large, a storage subsystem designer might be inclined to incorporate cache-awareness if the implementation effort is not unreasonable.

On the other hand, the system-level model predicts that cache-awareness consistently hurts storage subsystem

performance. The average response time increases by 21 percent for a scaling factor of one and by 17 percent for a scaling factor of two. The conclusion, given these performance predictions, is that SPTF is superior to SPCTF for these two workloads.[3]

The SPCTF algorithm hurts performance (for this workload) because of complex interactions between performance/workload feedback effects and the disk's prefetching behavior. Most of the read requests in *compress* are caused by sequential file accesses and are, therefore, largely (but not entirely) sequential. These requests are generated in a fairly regular manner. The task-executing process reads a file block, works with it, and then reads the next. The file system prefetches file block N + 1 when the process accesses (or attempts to access) block N. So, the number of pending read requests ranges between zero and two, greatly diminishing the importance of SPTF's difficulty with sets of pending sequential reads. Write requests, arriving in bursts when the syncer daemon awakens, occasionally compete with these read requests. The average response time is largely determined by how well the storage subsystem deals with these bursts. At some point during such a burst, there will be zero pending read requests and the disk scheduler (either variant) will initiate one of the write requests.

From this point, SPTF and SPCTF progress differently. SPTF will continue to service the write requests, independent of read request arrivals. The writes (like the reads) generally exhibit spatial locality with other writes and will, therefore, incur smaller positioning delays. So, any new read request(s) will wait for all of the writes to complete and will then be serviced in sequence as they arrive. On the other hand, SPCTF will immediately service a new read request if it hits in the on-board disk cache. At this point, the HP C2247 disk drive will begin to reposition the read/write head in order to prefetch additional data. After servicing the read request, SPCTF will initiate one of the pending writes. The disk will discontinue prefetching at this point (if it even reached the read's cylinder) and reposition to service the write. Frequently, the disk will not succeed in prefetching any data because repositioning the read/write head requires more time than it takes to service a cache hit and begin the next request. This cycle (read hit, failed prefetch, write request) may repeat several times before a new read request misses in the disk's cache. At this point, SPCTF behaves like SPTF, servicing the remainder of the writes before handling any reads. The time wasted repositioning the read/write head for aborted prefetch activity decreases disk efficiency and increases the average response time.

Given that initiating prefetch after a cache hit causes this performance problem, one might consider not doing so. However, this policy does improve performance and should remain in place. For example, for *compress* (unscaled), the system-level model predicts that SPTF performance drops by 3 percent when prefetch activity is not initiated after cache hits. Without such prefetch activity, the system-level model also predicts that SPCTF outperforms SPTF by 1 percent, which is not enough to compensate for the elimination of cache hit prefetching. Among the crossproducts of these options, the best storage subsystem performance (for this workload) is offered by SPTF scheduling and aggressive prefetching after cache hits.

The interactions that cause cache-awareness to hurt performance are obscure and easy to overlook without some indication that they represent a problem. Reference [26] compared SPCTF and SPTF, using extensive open subsystem simulations driven by disk request traces captured from commercial computer systems, and found SPCTF to be superior. None of the results suggested that SPCTF is ever inferior to SPTF. While the data presented above do not disprove the previous results (because the workloads are different), they certainly cast doubt. This example demonstrates that trivializing performance/workload feedback effects can lead to erroneous conclusions in nonobvious ways.

7 Criticality-Based Disk Scheduling

Even when storage subsystem performance is correctly predicted, the conventional subsystem-oriented methodology can promote suboptimal designs because commonly used subsystem performance metrics (e.g., request response times) do not always correlate with overall system performance metrics (e.g., task elapsed times). We illustrate this problem with a storage subsystem modification (prioritizing disk requests based on request criticality) that improves overall system performance while reducing performance as measured by subsystem metrics.

From a short-term viewpoint, time-critical and time-limited are clearly more important to system performance than time-noncritical requests. So, we modify the C-LOOK algorithm to maintain two distinct lists of pending requests and service requests from the higher-priority list first.[4] Time-critical and time-limited requests are placed in the high-priority list, and time-noncritical requests are placed in the low-priority list. Time-limited requests are grouped with time-critical requests because measurements indicate that they often have short time limits.

To implement the modified algorithm, a disk request scheduler must have per-request criticality information. Fortunately, request-generating system components (e.g., file systems and virtual memory managers) generally know a request's criticality class when it is generated. Requests caused by demand fetches and synchronous writes are time-critical. Background flushes initiated by the syncer daemon are time-noncritical. Requests resulting from prefetches and asynchronous metadata updates are usually time-limited. (In our implementation, we assume that all such requests are time-limited.) We modified the appropriate modules to pass this information to the device driver with each request as part of the I/O control block.

Fig. 3 evaluates criticality-based disk scheduling with commonly utilized performance metrics for the overall

3. For the example in the next section, increased request response times do not translate into increased elapsed times for tasks. In this example, they do.

4. Many variations and more aggressive algorithms can be devised, but our goal is not to fully explore the design space of disk scheduling algorithms that use request criticality information. Rather, our intent is to illustrate a problem with standalone storage subsystem models and indicate the potential of this performance enhancement.

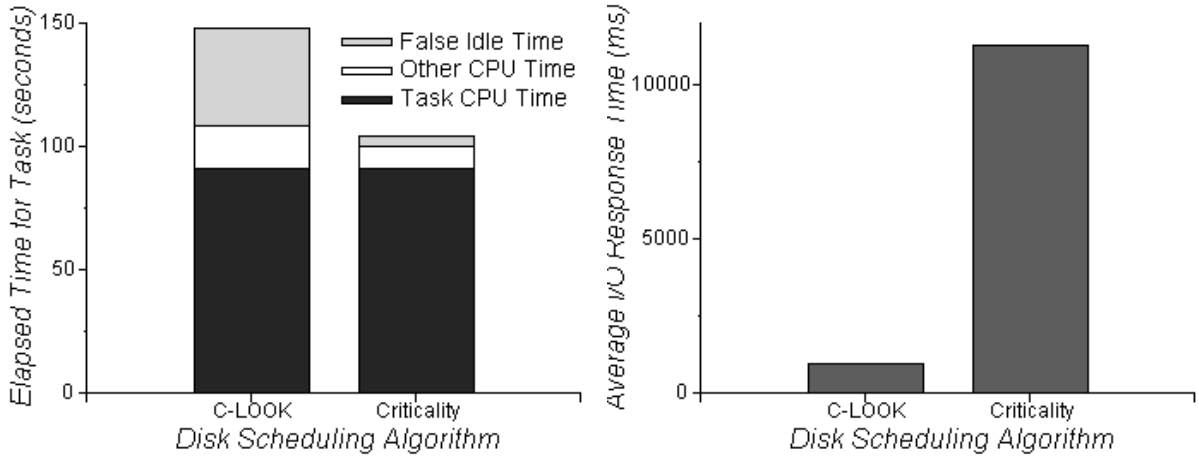


Fig. 3. Criticality-based scheduling of the *uncompress* workload. (a) Task elapsed time. (b) I/O response time.

system and the storage subsystem. Graph (a) shows overall system performance as measured by the time from when the task-executing process begins to when it exits (i.e., completes).[5] The elapsed times are broken into three regions: false idle time, CPU time used by the task-executing process, and other CPU time. The task computation time is independent of storage subsystem performance and, therefore, is not affected by changes to the disk request scheduler. The other CPU time consists mainly of system software execution by interrupt handlers and system daemons. Some of this activity competes with the task-executing process for CPU time and some of it occurs while the process is blocked waiting for I/O, thereby reducing false idle time. Graph (b) shows the average response time across all I/O requests, including those initiated in the background after the task-executing process completes. Table 6 provides additional information to assist in understanding the data in the figures.

For the *uncompress* workload, overall system performance increases by 30 percent when the disk scheduler uses request criticality information. At the same time, storage subsystem performance decreases by more than an order of magnitude. Most of the overall system performance improvement comes from reducing false idle time (by 90 percent) by expediting the completion of requests that cause processes to block. The remainder comes from a 45 percent reduction in other CPU time during the task's lifetime. (This activity still occurs after the task-executing process exits.) Storage subsystem performance decreases both because the scheduler focuses its efforts on system needs, rather than mechanical delays, and because processes progress more quickly and, therefore, generate I/O requests more quickly. The same qualitative results were observed for all workloads studied in [3].

This example demonstrates that storage subsystem performance metrics do not, in general, correlate with overall system performance metrics. Because storage subsystem performance decreases, evaluating criticality-based scheduling with subsystem metrics would lead one to dismiss it as poor. In fact, a recent publication observed reductions in subsystem performance for a similar algorithm (giving priority to reads over writes) and concluded that it is a bad

TABLE 6
CRITICALITY-BASED SCHEDULING OF THE *uncompress* WORKLOAD

Performance Metric	C-LOOK	Criticality
Elapsed Time for Task	147.5 sec	103.8 sec
Task Computation Time	90.3 sec	90.3 sec
Other CPU Time	17.6 sec	9.6 sec
Task I/O Wait Time	51.0 sec	4.3 sec
False Idle Time	39.6 sec	3.9 sec
Time-Critical Requests	153	153
Avg. Response Time	62.7 ms	29.1 ms
Max. Response Time	6,165 ms	95.1 ms
Time-Limited Requests	2,693	2,693
Avg. Response Time	42.5 ms	19.3 ms
Max. Response Time	11,000 ms	73.6 ms
Avg. Time Limit	31.0 ms	33.8 ms
% Satisfied in Time	95.9 %	86.5 %
Time-Noncritical Reqs.	7,989	7,989
Avg. Response Time	1,207 ms	15,300 ms
Max. Response Time	10,500 ms	69,700 ms
Total Request Count	10,835	10,835
Avg. Response Time	903.4 ms	11,256 ms
Avg. Service Time	8.0 ms	12.1 ms
% Disk Buffer Hits	22.4 %	8.2 %
Avg. Seek Distance	31 cyls	217 cyls
Avg. Seek Time	1.5 ms	4.2 ms

5. When the process exits, there are often dirty file cache blocks that remain to be flushed. While certainly important, the background write I/O requests for these dirty blocks are not part of the task completion time observed by a user. They can, however, interfere with the file and I/O activity of subsequent tasks if there is insufficient idle time between tasks. The last disk write completes at roughly the same time independent of the scheduling algorithm because of the periodic flush policy employed by the file system's syncer daemon.

design point [33]. Additional results, available in [3], show that simply excluding time-noncritical requests from the average response time is not sufficient because of the performance effects of time limits. We believe that overall system metrics must be produced directly (e.g., with a system-level model) rather than inferred from subsystem metrics.

8 Conclusions and Future Work

The conventional design-stage I/O subsystem performance evaluation methodology is too narrow in scope. Because standalone subsystem models ignore differences in how individual request response times affect system behavior, they can lead to erroneous conclusions. As a consequence, many previous results must be viewed with skepticism until they are verified either in real environments or with a more appropriate methodology.

Conventional methodology fails to accurately model feedback effects between I/O subsystem performance and the workload. Open subsystem models completely ignore feedback effects. As a result, open subsystem models tend to overestimate performance changes and allow unrealistic concurrency. When performance decreases, prediction error grows rapidly as the lack of feedback quickly causes saturation. When I/O subsystem performance increases, performance prediction errors of up to 30 percent are observed. Closed subsystem models assume unqualified feedback, generating a new request to replace each completed request. As a result, closed subsystem models tend to underestimate performance changes and completely ignore burstiness in request arrival patterns. Closed subsystem models rarely correlate with real workloads, leading to performance prediction errors as large as an order of magnitude.

Conventional methodology also relies upon I/O subsystem metrics, such as the mean request response time. These metrics do not always correlate well with overall system performance metrics, such as the mean elapsed time for user tasks. For example, the use of request criticality by the disk scheduler reduces elapsed times for user tasks by up to 30 percent, while concurrently increasing the mean I/O response time by as much as two orders of magnitude.

A new methodology based on system-level models is proposed and shown to enable accurate predictions of both subsystem and overall system performance. A simulation infrastructure that implements the proposed methodology is described and validated. The system-level simulation model's performance predictions match measurements of a real system very closely (within 5 percent in all comparisons).

Given the problems associated with conventional methodology, the obvious next step is to reevaluate previous storage subsystem research using the proposed methodology. This will require libraries of system-level traces collected from real user environments. Our instrumentation limits the length of system-level traces to the capacity of a dedicated kernel memory buffer. By partitioning the trace buffer into two or more sections and collecting trace data in one section as others are being copied to disk, much longer traces can be collected. Because all system activity is traced, the activity (CPU, memory, and I/O) related to trace collection can be identified and removed from the trace. Trace-collection overhead would, therefore, impact the traced workload only if users perceive the performance degradation and change their behavior. Sensitivity studies would be needed, but this approach may enable acquisition of very long system-level traces.

Criticality-based disk scheduling also appears promising. More thorough investigation is needed to understand the issues involved with more sophisticated algorithms, main memory saturation, and the reliability of new data. Some of these factors are explored in [3], [14].

Acknowledgments

We thank John Wilkes and Bruce Worthington for directly helping to improve the quality of this paper. During this I/O subsystem research, our group was very fortunate to have the financial and technical support of several industrial partners, including NCR, DEC, HaL, Hewlett-Packard, Intel, Motorola, MTI, and SES. In particular, NCR enabled this research with their extremely generous equipment gifts and by allowing us to generate instrumented operating system kernels with their source code. The disk used in the experiments was provided by Hewlett-Packard.

References

[1] G. Ganger and Y. Patt, "The Process-Flow Model: Examing I/O Performance from the System's Point of View," *Proc. ACM Sigmetrics Conf.*, pp. 86-97, May 1993.

[2] C. Ruemmler and J. Wilkes, "UNIX Disk Access Patterns," *Proc. 1993 Winter USENIX*, pp. 405-420, Jan. 1993.

[3] G. Ganger, "System-Oriented Evaluation of I/O Subsystem Performance," PhD dissertation, CSE-TR-243-95, Univ. of Michigan, Ann Arbor, June 1995.

[4] J. Ousterhout, "Why Aren't Operating Systems Getting Faster as Fast as Hardware?" *Proc. 1990 Summer USENIX*, pp. 247-256, June 1990.

[5] S. Carson and S. Setia, "Analysis of the Periodic Update Write Policy for Disk Cache," *IEEE Trans. Software Eng.*, vol. 18, no. 1, pp. 44-54, Jan. 1992.

[6] R.H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," *Proc. 15th ACM Symp. Operating Systems Principles*, pp. 79=95, Dec. 1995.

[7] P. Cao, E.W. Felten, and K. Li, "Implementation and Performance of Application-Controlled File Caching," *Proc. First USENIX Symp. Operating Systems Design and Implementation (OSDI)*, pp. 165-178, Nov. 1994.

[8] J. Griffioen and R. Appleton, "Reducing File System Latency Using a Predictive Approach," *Proc. 1994 Summer USENIX*, pp. 197-207, June 1994.

[9] R. Abbott and H. Garcia-Molina, "Scheduling I/O Requests with Deadlines: A Performance Evaluation," *Proc. IEEE Real-Time Systems Symp.*, pp. 113-124, 1990.

[10] S. Chen, J. Kurose, J. Stankovic, and D. Towsley, "Performance Evaluation of Two New Disk Request Scheduling Algorithms for Real-Time Systems," *J. Real-Time Systems*, vol. 3, pp. 307-336, 1991.

[11] M. Carey, R. Jauhari, and M. Livny, "Priority in DBMS Resource Scheduling," *Proc. Int'l Conf. Very Large Data Bases*, pp. 397-410, 1989.

[12] "Small Computer System Interface-2," ANSI X3T9.2, Draft Revision 10k, Mar. 1993.

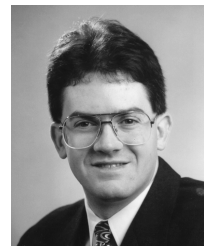
[13] R. Lary, storage architect, Digital Equipment Corp., personal communication, Feb. 1993.

[14] B. Worthington, "Aggressive Centralized and Distributed Scheduling of Disk Requests," PhD dissertation, CSE-TR-244-95, Univ. of Michigan, Ann Arbor, June 1995.

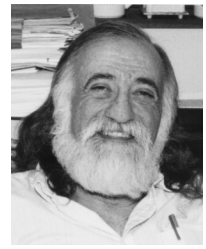
[15] P. Seaman and R. Soucy, "Simulating Operating Systems," *IBM Systems J.*, vol. 8, no. 4, pp. 264-279, 1969.

[16] W. Chiu and W. Chow, "A Performance Model of MVS," *IBM Systems J.*, vol. 17, no. 4, pp. 444-463, 1978.

[17] P. Haigh, "An Event Tracing Method for UNIX Performance Measurement," *Proc. Computer Measurement Group (CMG) Conf.*, pp. 603-609, 1990.
[18] K. Richardson and M. Flynn, "Time: Tools for Input/Output and Memory Evaluation," *Proc. Hawaii Int'l Conf. Systems Sciences*, pp. 58-66, Jan. 1992.
[19] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta, "Complete Computer System Simulation," *IEEE Parallel and Distributed Technology*, Fall 1995.
[20] J. Busch and A. Kondoff, "Disk Caching in the System Processing Units of the HP 3000 Family of Computers," *HP J.*, vol. 36, no. 2, pp. 21-39, Feb. 1985.
[21] E. Miller and R. Katz, "Input/Output Behavior of Supercomputing Applications," *Proc. Supercomputing '91*, pp. 567-576, 1991.
[22] A. Dan, D. Dias, and P. Yu, "Buffer Analysis for a Data Sharing Environment with Skewed Data Access," *IEEE Trans. Knowledge and Data Eng.*, vol. 6, no. 2, pp. 331-337, 1994.
[23] C. Thekkath, J. Wilkes, and E. Lazowska, "Techniques for File System Simulation," *Software—Practice and Experience*, vol. 24, no. 11, pp. 981-999, Nov. 1994.
[24] L. Mummert, M. Ebling, and M. Satyanarayanan, "Exploiting Weak Connectivity for Mobile File Access," *Proc. 15th ACM Symp. Operating Systems Principles*, pp. 143-155, Dec. 1995.
[25] Hewlett-Packard Company, "HP C2244/45/46/47 3.5-Inch SCSI-2 Disk Drive Technical Reference Manual," part no. 5960-8346, ed. 3, Sept. 1992.
[26] B. Worthington, G. Ganger, and Y. Patt, "Scheduling Algorithms for Modern Disk Drives," *Proc. ACM Sigmetrics Conf.*, pp. 241-251, May 1994.
[27] M.K. McKusick, W. Joy, S. Leffler, and R. Fabry, "A Fast File System for UNIX," *ACM Trans. Computer Systems*, vol. 2, no. 3, pp. 181-197, Aug. 1984.
[28] R. Gingell, J. Moran, and W. Shannon, "Virtual Memory Architecture in SunOS," *Proc. 1987 Summer USENIX*, pp. 81-94, June 1987.
[29] J. Moran, "SunOS Virtual Memory Implementation," *Proc. EUUG Conf.*, pp. 285-300, Spring 1988.
[30] C. Ruemmler and J. Wilkes, "An Introduction to Disk Drive Modeling," *Computer*, vol. 27, no. 3, pp. 17-28, Mar. 1994.
[31] M. Seltzer, P. Chen, and J. Ousterhout, "Disk Scheduling Revisited," *Proc. 1990 Summer USENIX*, pp. 313-324, Jan. 1990.
[32] D. Jacobson and J. Wilkes, "Disk Scheduling Algorithms Based on Rotational Position," Technical Report HPL-CSP-91-7, Hewlett-Packard, Feb. 1991.
[33] R. Geist and J. Westall, "Disk Scheduling in Linux," *Proc. Computer Measurement Group (CMG)*, pp. 739-746, Dec. 1994.



Gregory R. Ganger received a BS (1991) in computer science and an MS (1993) and PhD (1995) in computer science and engineering from the University of Michigan. He is an assistant professor of electrical and computer engineering and computer science at Carnegie Mellon University. He has broad research interests in computer systems, including operating systems, networking, storage systems, architecture and distributed systems. Dr. Ganger spent 2.5 years as a postdoctoral associate at the MIT Lab for Computer Science before moving to Carnegie Mellon University. He is a member of the IEEE Computer Society.



Yale N. Patt received a BS in electrical engineering from Northeastern University and an MS and PhD in electrical engineering from Stanford University. He is a professor of electrical engineering and computer science at the University of Michigan. His research interests include high-performance computer architecture, processor design, and computer systems implementation. Dr. Patt is the recipient of the 1996 ACM/IEEE Eckert-Mauchly Award and the 1995 IEEE Emanuel R. Piore Aware. He is a fellow of the IEEE.