Published as Report CSE-TR-243-95, Department of EECS, University of Michigan, Ann Arbor, June 1995.

System-Oriented Evaluation of I/O Subsystem Performance

by

Gregory Robert Ganger

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1995

Doctoral Committee:
Professor Yale N. Patt, Chair
Professor Peter M. Banks
Professor Edward S. Davidson
Professor Trevor N. Mudge
Professor Joseph Pasquale, University of California, San Diego

ABSTRACT

System-Oriented Evaluation of I/O Subsystem Performance

by
Gregory Robert Ganger

Chair: Yale N. Patt

This dissertation demonstrates that the conventional approach for evaluating the performance of an I/O subsystem design, which is based on standalone subsystem models, is too narrow in scope. In particular, conventional methodology treats all I/O requests equally, ignoring differences in how individual request response times affect system behavior. As a result, it often leads to inaccurate performance predictions and can thereby lead to incorrect conclusions and poor design choices. A new methodology, which expands the model's scope to include other important system components (e.g., CPUs and system software), is proposed and shown to enable accurate predictions of both subsystem and overall system performance.

This dissertation focuses on two specific problems with conventional methodology:

1. Benchmark workloads 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 of requests (in particular, subsequent request arrivals).

2. Changes in I/O subsystem performance (e.g., as measured by mean request response times) do not always translate into similar changes in overall system performance (e.g., as measured by mean elapsed times for user tasks).

These problems are fundamental to the subsystem-oriented approach and are independent of the model's accuracy. The first problem is illustrated with several examples where commonly-utilized workload generators trivialize feedback effects and produce unrealistic workloads. In each case, quantitative and/or qualitative errors result. The second problem is illustrated with a disk scheduling algorithm that gives priority to those requests that are most critical to overall system performance. This change increases overall system performance while decreasing storage subsystem performance (as indicated by subsystem metrics). In all of the experiments, the new methodology is shown to avoid the short-comings of conventional methodology.

To Jennifer, for your extraordinary patience, love and support.

ACKNOWLEDGEMENTS

I could not have survived the Ph.D. process without the guidance, friendship and support of many people.

I will always be grateful to Yale Patt for all that he has done for me during my years at the University of Michigan. Yale poured uncountable hours into my development, and his guidance has been invaluable.

The many hours at the office were made both bearable and enjoyable by the other members of the research group, including Mike Butler, Po-Yung Chang, Chris Eberly, Carlos Fuentes, Eric Hao, Robert Hou, Lea-Hwang Lee, Dennis Marsa, Eric Sprangle, Jared Stark, and Tse-Yu Yeh. Two, in particular, kept me sane over the years. During my early years of graduate school, Mike Butler empathized with my various complaints and calmly informed me that the worst was yet to come. Bruce Worthington, who has been my research partner for the past couple of years, is a godsend. He always seems to come up with practical solutions to difficult problems. I will always value his insights and friendship.

My research and personal development has been enhanced by several industry research internships, and many people at each of the companies have been good friends and colleagues. During internships at NCR, Jim Browning, Chas Gimarc, Mike Leonard and Rusty Ransford shared many insights and helped me to understand their systems, which became the experimental platforms for much of my research. During my internship at DEC, Bruce Filgate, Fran Habib, Rama Karedla, Doug Sharp and Peter Yakutis patiently weaned me from academic ideals and exposed me to the frightening reality of the storage subsystem marketplace. In two intense months at HP Labs, Richard Golding, Tim Sullivan, Carl Staelin and John Wilkes shared a wide variety of ideas and allowed me to participate in their unique research environment.

In addition to those mentioned above, I would like to thank Peter Chen, Garth Gibson, Peter Honeyman, Dave Hudak, Richie Lary, Dave Nagle, Stuart Sechrest, and Rich Uhlig for their insights on my research problems. I am particularly grateful to Peter Banks, Ed Davidson, Trevor Mudge and Joe Pasquale for making time in their extremely busy schedules to be members of my doctoral committee. Also, I thank Richard Golding, John Wilkes, and Bruce Worthington for reading sections of this dissertation and providing feedback.

Jimmy Pike of NCR Corporation (now AT&T/GIS) provided the initial support for our group's storage subsystem research. NCR provided most of the funding, equipment and operating system source code for my research. Hewlett-Packard provided workstations, disk drives and disk request traces for our research group. Digital Equipment Corporation provided workstations and disk request traces.

I thank the Computer Measurement Group for awarding me one of their fellowships, which partially funded my final year of graduate school.

Several staff members at the University of Michigan have kept things from flying apart on a day-to-day basis. Michelle Chapman, our group's administrative assistant, has consistently been helpful above and beyond the call of duty. The other ACAL staff, Paula Denton, Denise DuPrie and Jeanne Patterson, have also been very helpful. The staff members of the departmental computing organization kept the main servers running smoothly and always went out of their way to provide assistance.

I am extremely grateful to my family, especially my wife and parents, for their unconditional love, for supporting my decision to stay in graduate school at Michigan, and for accepting the fact that I was almost always preoccupied with work.

Finally, a global thank you to all of the CSE faculty and students at the University of Michigan who stopped by my open office door for chats about whatever.

TABLE OF CONTENTS

CHAPTERS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

LIST OF APPENDICES

APPENDIX

CHAPTER 1

Introduction

1.1 The Problem

The performance of the input/output (I/O) subsystem plays a large role in determining overall system performance in many environments. The relative importance of this role has increased steadily for the past 25 years and should continue to do so for two reasons. First, the components that comprise the I/O subsystem have improved at a much slower rate than other system components. For example, microprocessor performance grows at a rate of 35–50 percent per year [Myers86], while disk drive performance grows at only 5–20 percent per year [Lee93]. As this trend continues, applications that utilize any quantity of I/O will become more and more limited by the I/O subsystem [Amdahl67]. Second, advances in technology enable new applications and expansions of existing applications, many of which rely on increased I/O capability.

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, such as the mean request response time. 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 (the first factor) can be achieved by careful calibration against one or more real I/O subsystems. Within the context of the conventional methodology, the latter two factors are less well-understood and are the focus of this dissertation.

1.2 Thesis Statement

I contend that the conventional methodology for evaluating the performance of I/O subsystem designs, which focuses on I/O subsystem models in isolation, is too narrow in scope In particular, conventional methodology treats all I/O requests equally, ignoring differences in how individual response times affect system behavior. As a result, it often leads to inaccurate performance predictions and can thereby lead to incorrect conclusions and poor designs. This dissertation proposes a new methodology, based on **system-level models**, that directly solves the problems with the conventional methodology.

1.3 Overview of Dissertation

The conventional methodology is flawed in two important ways:

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 incoming workload of I/O requests (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 the subsystem-oriented approach and are independent of the model's accuracy. The proposed system-level modeling methodology directly solves both of these problems.

Both problems arise because the conventional methodology tends to treat all I/O requests as equally important. The foundation for the thesis is provided by describing three distinct classes of **request criticality** based on how individual requests affect overall system performance. Generally speaking, one request is more **critical** than another if it is more likely to block application processes and thereby waste CPU cycles. Most I/O workloads consist of a mixture of requests from the three classes. The common approaches to workload generation fail to accurately recreate the effects of request criticality mixtures.

While the thesis of this dissertation applies 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, this dissertation focuses on the secondary storage subsystem. The conventional approach to performance evaluation of storage subsystem designs is described in detail. Also, previous storage subsystem research is briefly reviewed to establish that it is commonly utilized.

A detailed system-level simulation model is described, providing an existence proof for the feasibility of the proposed methodology. This simulator satisfies the three requirements of a good performance evaluation methodology outlined above. (1) Extensive validation establishes the model's accuracy. (2) The system-level model correctly incorporates the complex feedback effects that destroy the representativeness of most subsystem model workloads. Workload representativeness is an exercise in choosing the appropriate benchmarks. (3) The system-level model reports overall system performance metrics directly.

The system-level simulation model and a detailed storage subsystem simulation model are used to provide several concrete examples where the conventional methodology produces incorrect conclusions (both quantitative and qualitative) regarding subsystem performance. These examples represent evidence of the first problem with standalone I/O subsystem models, that is, the misrepresentation of feedback effects in the workload. A good system-level model removes this problem by expanding the scope of the model, making the feedback effects part of the model rather than an aspect of the input workload.

The system-level simulation model is used to provide a concrete example where the conventional methodology promotes an I/O subsystem design that is poor for overall system performance. This occurs when, independent of the accuracy of the performance predictions, I/O subsystem performance improvements fail to translate into overall system performance. In an exploration of criticality-based disk request scheduling, I show that overall system performance increases while storage subsystem performance decreases. This demonstrates the existence of the second of the two problems with standalone I/O subsystem models. A good system-level model solves this problem by providing overall system performance metrics as well as storage subsystem metrics.

1.4 Contributions

This dissertation makes three main contributions:

- It is shown that the conventional I/O subsystem performance evaluation methodology is fundamentally flawed and can lead to incorrect conclusions and sub-optimal designs.
- The concept of request criticality, which distinguishes I/O requests based on how they interact with application processes, is identified and defined. This is an important step towards understanding the relationship between I/O performance and system performance.
- It is shown that a detailed, validated system-level simulation model can accurately predict both subsystem and system performance changes. The proposed methodology, which is based on system-level models, directly solves the fundamental problems with conventional methodology.

The tools described in this dissertation also represent useful, although secondary, contributions.

1.5 The Organization

The remainder of the dissertation is organized as follows. Chapter 2 introduces the concept of request criticality and explains how it relates to the fundamental problems of conventional methodology. Chapter 3 describes previous work related to request criticality and system-level models. Also, the conventional methodology is described in detail. Chapter 4 describes the proposed methodology, including the system-level models on which it is founded. Chapter 5 describes and validates the simulation infrastructure used in this dissertation. Chapter 6 proves the existence of the first problem outlined above with several concrete examples where the conventional methodology leads to incorrect conclusions (quantitative and qualitative) regarding storage subsystem performance by failing to properly incorporate interactions between request completions and subsequent arrivals. Chapter 7 investigates criticality-based disk scheduling. The results illustrate the existence of the second problem outlined above by providing an example where system performance increases while storage subsystem performance decreases. Chapter 8 summarizes the contributions of this dissertation and suggests some avenues for future research.

CHAPTER 2

Request Criticality

The two fundamental problems with conventional methodology are caused by variations in how individual I/O requests interact with the rest of the system. Performance/workload feedback effects are complicated by these variations, making commonly utilized subsystem workload generators inappropriate. Also, different requests affect overall system performance in different ways and to different degrees, invalidating subsystem metrics that treat all requests as equally important. This chapter defines three distinct classes of request criticality to help explain these variations. Common subsystem workload generators are described in terms of the three classes.

Storage accesses interfere with system performance in several ways. Some, such as increased system bus and memory bank contention, depend mainly on the quantity and timing of the accesses and are essentially independent of criticality. Others, such as false idle time and false computation time, are highly dependent on request criticality. **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 differentiated from regular idle time, which is due to a lack of available work in the system. **False computation time** denotes time that is wasted handling a process that blocks and waits for an I/O request to complete. This includes the time required to disable the process, context switch to a new process and, upon request completion, re-enable the process. False computation time also includes the various cache fill penalties associated with unwanted context switches.

2.1 Three Classes of Request Criticality

Request criticality refers to how a request's response time (i.e., the time from issue to completion) affects system performance and, in particular, process wait times. I/O requests separate into three classes: time-critical, time-limited and time-noncritical. This taxonomy is based upon the interaction between the I/O request and executing processes.

Time-Critical

A request is **time-critical** if the process that generates it must stop executing until the request is complete. Examples of time-critical requests include demand page faults, synchronous file system writes and database block reads.

Time-critical requests, by definition, cause the processes that initiate them to block and wait until they complete. In addition to false computation time, false idle time is accumulated if there are no other processes that can be executed when the current process blocks. This is certainly the largest concern, as it completely wastes the CPU for some period of time (independent of the processor speed) rather than some number of cycles. To reduce false idle time, time-critical requests should be expedited.

Time-Limited

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 time-limited requests.

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 the performance problems described above. 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.

Time-Noncritical

No process waits for **time-noncritical** requests. They must be completed to maintain the accuracy of the non-volatile storage, to free the resources (e.g., memory) that are held on their behalf, and/or to allow some background activity to progress. Examples of time-noncritical requests are delayed file system writes, requests issued for background data re-organization, and database updates for which the log entry has been written.

In reality, there are no truly time-noncritical requests. If background activity is never handled, 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 these 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).

Except when main memory saturates, time-noncritical requests impact performance indirectly. They can interfere with the completion of time-limited and time-critical requests, causing additional false idle time and false computation time. Delays in completing time-noncritical requests can also reduce the effectiveness of the in-memory disk cache.

Time-noncritical requests can also have completion time requirements if the guarantees offered by the system require that written data reach stable storage within a specified amount of time. These requests are not time-limited according to my 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.

2.2 Relation to Workload Generators

To clarify the request taxonomy and show how conventional methodology fails to adequately deal with criticality mixtures, the two common workload generation approaches (open and closed) are described in terms of the request criticality classes. The workload generation approaches are described in more detail in section 3.3.3.

Open subsystem models use predetermined arrival times for requests. They assume that the workload consists exclusively of time-noncritical requests. That is, changes to the completion times of I/O requests have no effect on the generation of subsequent requests.

Closed subsystem models maintain a constant population of requests. Whenever completion is reported for a request, a new request is generated, delayed by some think time and issued into the storage subsystem. Therefore, a closed subsystem model assumes that the workload consists exclusively of time-critical requests. In most closed models, the think time between I/O requests is assumed to be zero so that there are a constant number of outstanding requests.

Most real workloads are neither of these extremes. Far more common than either is a mixture of time-critical, time-limited and time-noncritical requests. For example, extensive measurements of three different UNIX systems ([Ruemmler93]) showed that time-critical requests ranged from 51–74 percent of the total workload, time-limited ranged from 4–8 percent and time-noncritical ranged from 19–43 percent. Because of the complex request criticality mixtures found in real workloads, standalone storage subsystem models (regardless of how accurately they emulate real storage subsystems) often produce erroneous results. This problem exists because of the trivialized feedback effects assumed by the simple workload generators that are commonly utilized. Also, variations in how individual I/O requests affect overall system performance make it infeasible to predict (in general) system performance changes with storage subsystem metrics.

[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.

CHAPTER 3

Previous Work

The purpose of this chapter is to motivate and provide a context for the work presented in this dissertation. Previous work relating both to request criticality and to system-level models is described. Storage subsystem models, the tools that form the basis of the conventional methodology for evaluating the performance of a storage subsystem design, are described. Popular workload generators for such models are also described. These workload generators are at the root of the conventional methodology's short-comings because they trivialize feedback effects between storage subsystem performance and system behavior. A brief survey of previous storage subsystem research demonstrates that, despite its short-comings, storage subsystem modeling is commonly utilized.

3.1 Request Criticality

Although I have found no previous work which 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 non-synchronous (i.e., time-limited and time-noncritical) [Ousterhout90, McVoy91, Ruemmler93]. In their extensive traces of disk activity, Ruemmler and Wilkes captured information (as flagged by the file system) indicating whether or not each request was synchronous. They found that 50-75% of disk requests are synchronous, largely due to the write-through metadata cache on the systems traced.

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) [Carson92, Mogul94]. 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 gave 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.

When disk blocks are cached in a non-volatile memory, most write requests from the cache to the disk are time-noncritical. In such environments, the cache should be designed to minimize read response times while ensuring that the cache does not fill with dirty blocks [Reddy92, Biswas93, Treiber94]. With non-volatile cache memory becoming more and more common, it becomes 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 prefetching activity (e.g., [Patterson93, Griffioen94, Cao95]), hoping to utilize high-throughput storage systems to reduce read latencies.

Priority-based disk scheduling has been examined and shown to improve system performance. For example, [Carey89] evaluates a priority-based SCAN algorithm where the priorities are assigned based on the process that generates the request. Priority-based algorithms have also been studied in the context of real-time systems, with each request using the deadline of the task that generates it as its own. [Abbott90] describes the FD-SCAN algorithm, wherein the SCAN direction is chosen based on the relative position of the pending request with the earliest feasible deadline. [Chen91a] describes two deadline-weighted Shortest-Seek-Time-First algorithms and shows that they provide lower transaction loss ratios than non-priority algorithms and the other priority-based algorithms described above. In all of these cases, the priorities assigned to each request reflects the priority or the deadline of the associated processes rather than criticality.

Finally, the *Head-Of-Queue* [SCSI93] or *express* [Lary93] 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 [Worthington95a].

3.2 System-Level Modeling

This dissertation, in part, proposes the use of system-level models for evaluating I/O subsystem designs. This section describes previous work relating to system-level models and their use in storage subsystem performance evaluation.

[Seaman69] and [Chiu78] describe system-level modeling efforts used mainly for examining alternative system configurations (as opposed to I/O subsystem designs). [Haigh90] 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 my trace acquisition tool. [Richardson92] describes a set of tools under development that are intended to allow for studying I/O performance as part of the entire system. These tools are based on instruction-level traces. While certainly the ideal case (i.e., simulating the entire activity of the system is more accurate than abstracting part of it away), it is not practical. The enormous simulation times and instruction trace storage requirements, as well as the need for instruction level traces of operating system functionality, make this approach both time- and cost-prohibitive.

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. For example, [Miller91] uses a simple system-level model to study the effects of read-ahead and write buffering on supercomputer applications. [Busch85] examines the transaction processing performance impact of changes to hit ratios and flush policies for disk block caches located in main memory. [Dan94] studies transaction throughput as a function of the database buffer pool size for skewed access patterns. My work extends these approaches in two ways: (1) by using a thorough, validated system-level model, and (2) by using system-level models to evaluate storage subsystem designs in addition to host system cache designs.

An interesting technique for replaying file system request traces in a realistic manner has recently been introduced and used to evaluate reintegration policies for disconnected and weakly connected distributed file systems [Mummert95]. Each event in their traces contains a file system request and an associated user identification. A trace is re-organized to consist of per-user sequences of file system requests. The replay process issues requests for each sequence in a closed-loop fashion. The measured inter-request time is used if it exceeds a think threshold parameter, λ, and ignored (i.e., replaced with zero) otherwise. The think threshold's role is to distinguish between user think times, which arguably have not changed much in the past 50 years, and job computation times, which continue to improve dramatically with time. Sensitivity analyses are still needed to show that this approach works and to identify appropriate values for λ. Mummert, et al., selected λ equal to 1 second and 10 seconds. [Thekkath94] promotes a similar file system trace replay approach with λ set to zero. 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, this technique does offer a healthy supply of input workloads for system-level models, which would of course need a module that simulates file system functionality [Thekkath94].

3.3 Conventional Methodology

The conventional approach to evaluating the performance of a storage subsystem design is to construct a model (analytic or simulation) of the components of interest, exercise the model with a sequence of storage I/O requests, and measure performance in terms of response times and/or throughput. This section describes the conventional methodology in detail, including a general view of storage subsystem models and a discussion of popular components and the varying levels of detail with which they may be modeled. Common performance metrics are briefly described. Common approaches to workload generation are described with special attention given to those aspects that are at the root of the shortcomings of the conventional methodology. A number of examples from the open literature where subsystem models have been used to investigate storage subsystem design issues are provided.

3.3.1 Storage Subsystem Models

A **storage subsystem model** consists of modules for one or more storage subsystem components and an interface to the rest of the computer system. Requests are issued to the model via the interface and are serviced in a manner that imitates the behavior of the corresponding storage subsystem. The components comprising a particular model, as well as the level of detail with which each is simulated, should depend upon the purpose of the model. Generally speaking, more components and/or finer detail imply greater accuracy at the cost of increased development and execution time.

Figure 3.1 shows an example of a storage subsystem. This particular example includes disk drives, a small disk array, and intelligent cached I/O controller, a simple bus adapter, several buses, a device driver and an interface to the rest of the system. Each of these storage subsystem components is briefly described below, together with discussion of how they are commonly modeled.

Interface to Rest of System

The interface between the storage subsystem and the remainder of the system is very simple. Requests are issued by the system and completion is reported for each request when appropriate. A request is defined by five values:

- **Device Number**: the logical storage device to be accessed. The device number is from the system's viewpoint and may be remapped several times by different components as it is routed (through the model) to the final physical storage device. This field is unnecessary if there is only one device.

- **Starting Block Number**: the logical starting address to be accessed. The starting block number is from the system's viewpoint and may be remapped several times by different components as it is routed to the final physical storage device.

- **Size**: the number of bytes to be accessed.

- **Flags**: control bits that define the type of access requested and related characteristics. The most important request flag component indicates whether the request is a read or a write. Other possible components might indicate whether written data should be re-read (to verify correctness), whether a process will immediately block and wait for the request to complete, and whether completion can be reported before newly written data are safely in non-volatile storage.

- **Main Memory Address**: the physical starting address in main memory acting as the destination (or source). The main memory address may be represented by a vector of memory regions in systems that support gather/scatter I/O. This field is often not included in request traces and is only useful for extremely detailed simulators.

A detailed model may also emulate the DMA (Direct Memory Access) movement of data to and from host memory. Some storage subsystem simulators maintain an image of the data and modify/provide it as indicated by each request. For example, such support is

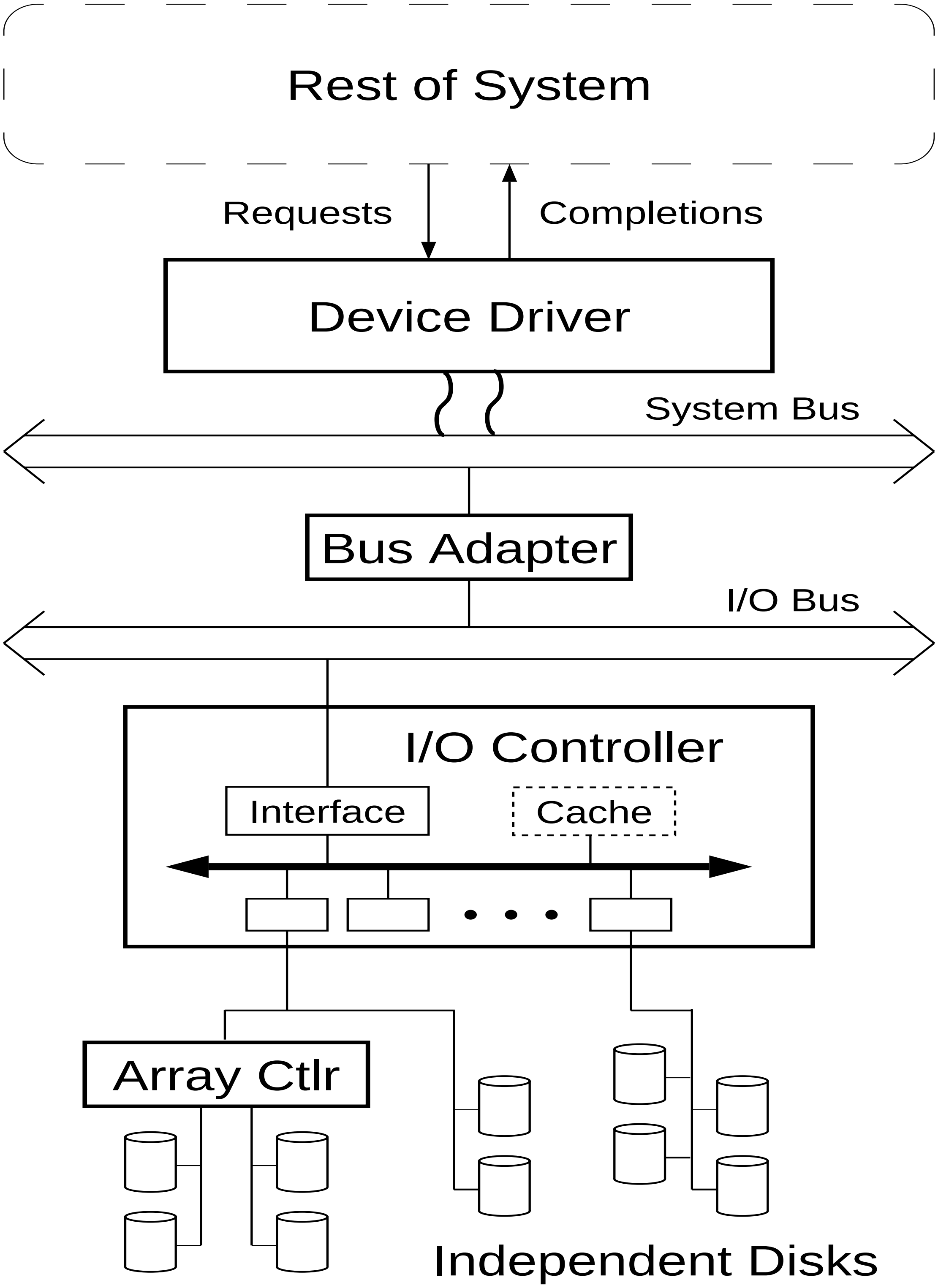


Figure 3.1: Block Diagram of a Storage Subsystem.

useful when the storage subsystem simulator is attached to a storage management simulator (e.g., a file system or database model) [Kotz94].

Device Driver

The device driver deals with device specific interactions (e.g., setting/reading controller registers to initiate actions or clear interrupts), isolating these details from the remainder of the system software. Device drivers will often re-order (i.e., schedule) and/or combine requests to improve performance. The device drivers are system software components that execute on the main computer system processors.

In a storage subsystem model, the device drivers (if included) interface with the rest of the system. In many models, they are represented as zero-latency resources that deal with request arrivals/completions and make disk request scheduling decisions. More aggressive simulators may include delays for the different device driver activities (e.g., request initiation and interrupt handling) so as to predict the CPU time used to handle storage I/O activity.

Buses

In a computer system, buses connect two or more components for communication purposes. To simplify discussion, the term "bus" will be used for point-to-point (possibly unidirectional) connections as well as connections shared by several components. The decision as to which component uses the bus at any point in time, or arbitration, may occur before each bus cycle (as is common for system buses and general I/O buses, such as MicroChannel [Muchmore89]) or less frequently (as is common for storage subsystem buses, such as SCSI [SCSI93]).

In most storage subsystem models, buses are modeled as shared resources with ownership characteristics that depend on the arbitration technique employed. That is, the bus may be exclusively owned by one component for some period of time or several distinct data movements may be interleaved. Depending on the level of detail, bus transmission times can depend on the quantity of data being moved, the maximum bus bandwidth and characteristics of the communicating components (e.g., buffer sizes and bus control logic).

Storage Controllers and Bus Adapters

In a computer system, storage controllers manage activity for attached components and bus adapters enable communication across buses. A storage controller, together with the attached devices, often acts as a separate storage subsystem with an interface similar to that for the overall subsystem. Such controllers generally contain CPUs and memory, execute a simplified operating system (referred to as firmware) and control the activity of several storage devices. Disk request scheduling, disk block caching, and disk array management are increasingly common functions performed by storage controller firmware. Bus adapters, on the other hand, generally consist of small amounts of buffer memory and very simple logic to deal with bus protocols.

In most storage subsystem models, bus adapters are modeled as zero-latency resources that simply allow data movement from one bus to another. Any latencies are generally assumed to be included in the associated bus transfer times. Very detailed models might model each individual bus cycle and the associated adapter activity in order to (for example) study optimal buffer sizes within the adapter. Storage controllers, on the other hand, are much more complex and can therefore be modeled in a much wider variety of ways. Simple controller models use a zero-latency resource to model various controller activities, including request processing, bus and device management, internal data movement, scheduling, caching and array management. More detailed models incorporate the CPU utilization, internal bus bandwidth and internal memory capacity used for the different activities.

Disk Drives

In most computer systems, the disk drive remains the secondary storage device of choice. Generally, all permanent data are written to disk locations and disk drives act as the backing store for the entire system. In some very large data storage systems, disk drives are used as caches for near-line tertiary devices (e.g., robotic tape libraries and optical disk jukeboxes). Disk drives have grown in complexity over the years, using more powerful CPUs and increased on-board memory capacity to augment improvements in mechanical components. Descriptions of disk drive characteristics can be found in appendix B and in [Ruemmler94, Worthington94].

In a storage subsystem model, disk drives can be simulated as resources with a wide variety of complexities, ranging from servers with delays drawn from a single probability distribution (e.g., constant or exponential) to self-contained storage systems with bus control and speed-matching, request queueing and scheduling, on-board disk block caching, CPU processing delays and accurate mechanical positioning delays. [Ruemmler94] examines several points in this range of options. Appendix B describes the options supported in the disk module of my simulation environment.

3.3.1.1 Detail and Accuracy

The level of detail in a storage subsystem model should depend largely upon the desired accuracy of the results. More detailed models are generally more accurate, but require more implementation effort and more computational power. Simple models can be constructed easily and used to produce "quick-and-dirty" answers. More precise performance studies, however, must use detailed, validated simulation models (or real implementations) to avoid erroneous conclusions. For example, [Holland92] uses a more detailed storage subsystem simulator to refute the results of [Muntz90] regarding the value of piggybacking rebuild requests on user requests to a failed disk in a RAID 5 array. As another example, [Worthington94] determines (using an extremely detailed disk simulator) that the relative performance of seek-reducing algorithms (e.g., Shortest-Seek-Time-First, V-SCAN(R) and C-LOOK) is often opposite the order indicated by recent studies [Geist87, Seltzer90, Jacobson91].[1]

It is worth reiterating that the problems addressed in this thesis are independent of how well a storage subsystem model imitates the corresponding real storage subsystem. Even the most accurate storage subsystem models suffer from two fundamental problems, because

[1] This discrepancy in results is also partly due to the non-representative synthetic workloads used in the older studies.

of the simplistic workload generators and narrowly-focused performance metrics. In fact, many prototypes and real storage subsystems have been evaluated with the same evaluation techniques (e.g., [Geist87a, Chen90a, Chervenak91, Chen93a, Geist94]).

3.3.2 Storage Performance Metrics

The three most commonly used storage subsystem performance metrics are request response times (averages, variances and/or distributions), maximum request throughputs and peak data bandwidth. The **response time** for a request is the time from when it is issued (i.e., enters the subsystem via the interface) to when completion is reported to the system. **Request throughput** is a measure of the number of requests completed per unit of time. **Data bandwidth** is a measure of the amount of data transferred per unit time. Secondary performance measures, such as disk cache hit rates, queue times, seek times and bus utilizations, are also used to help explain primary metric values.

3.3.3 Workload Generation

The workloads used in a performance study can be at least as important as model accuracy or performance metrics. The "goodness" of an input workload is how well it represents workloads generated by real systems operating in the user environments of interest. A quantified goodness metric should come from performance-oriented comparisons of systems under the test workload and the real workload [Ferr84]. The goal of this dissertation is not to examine the space of possible workloads or the range of reasonable goodness metrics, but to explore fundamental problems with the manner in which common workload generators for storage subsystem models trivialize important performance/workload feedback effects. In chapter 6, I utilize performance-oriented comparisons to show how these flaws can lead to both quantitative and qualitative errors regarding storage subsystem performance.

A storage subsystem workload consists of a set of requests and their arrival times (i.e., the times at which the system issues each request to the subsystem). With this definition, we have broken the workload down into two components, one spatial and one temporal. The spatial component deals with the five values described earlier that define a request. The temporal component deals with the time that a request arrives for service, which can depend partially on the response times of previous requests. The two components are of course related, and both are important. However, I will be focusing on the latter component here, as it lies at the root of the problems addressed in this dissertation.

Almost all model-based (analytic or simulation) performance studies of the storage subsystem can be divided into two categories (open and closed) based on how request arrival times are determined. The remainder of this subsection will describe these two categories, including how each allows for workload scaling. Workload scaling (i.e., increasing or decreasing the arrival rate of requests) is an important component of performance studies as it allows one to examine how a design will behave under a variety of situations.

Open Subsystem Models

Open subsystem models use predetermined arrival times for requests, independent of the storage subsystem's performance. So, an open subsystem model assumes there is no feedback between individual request response times and subsequent request arrival times. If the storage subsystem cannot handle the incoming workload, then the number of outstanding requests grows without bound. Open queueing models and most trace-driven simulations of the storage subsystem are examples of open subsystem models.

The central problem with open subsystem models is the assumption that there is no performance/workload feedback, ignoring real systems' tendency to regulate (indirectly) the storage workload based on storage performance. That is, when the storage subsystem performs poorly, the system will spend more time waiting for it (rather than generating additional work for it). One effect of this problem is that the workload generator for an open subsystem model may allow requests to be outstanding concurrently that would never in reality be outstanding at the same time (e.g., the read and write requests that comprise a read-modify-write action on some disk block).

The most common approach to workload scaling in an open subsystem model multiplies each inter-arrival time (i.e., the time between one request arrival and the next) by a constant scaling factor. For example, the workload can be doubled by halving each inter-arrival time. This approach to scaling tends to increase the unrealistic concurrency described above. To avoid this increase, [Treiber94] uses trace folding, wherein the trace is sliced into periods of time that are then interleaved. The length of each section should be long enough to prevent undesired overlapping, yet short enough to prevent loss of time-varying arrival rates. Treiber and Menon used 20 seconds as a convenient middle-ground length for each section. Trace folding should exhibit less unrealistic concurrency than simple trace scaling, but does nothing to prevent it.

Closed Subsystem Models

In a **closed subsystem model**, request arrival times depend entirely upon the completion times of previous requests. Closed subsystem models maintain a constant population of requests. Whenever completion is reported for a request, a new request is generated and issued into the storage subsystem.[2] That is, closed subsystem models assume unqualified feedback between storage subsystem performance and the incoming workload. Closed queueing models and simulations that maintain a constant number of outstanding requests are examples of closed subsystem models.

The main problem with closed subsystem models is that they ignore burstiness in the arrival stream. Measurements of real storage subsystem workloads have consistently shown that arrival patterns are bursty, consisting of occasional periods of intense activity inter-

[2] Requests in a closed model can spend think time in the system before being issued back into the subsystem model. For example, non-zero think times might be used to represent the processing of one block before accessing the next. While rare, non-zero think times have been used in published storage subsystem research. For example, [Salem86] uses a closed workload of <read block, process block, write block (optional)> sequences to emulate a generic file processing application.

spersed with long periods of idle time (i.e., no incoming requests). With a constant number of requests in the system, there is no burstiness.

Workload scaling in a closed subsystem model can be accomplished by simply increasing or decreasing the constant request population. If non-zero think times are used, workload scaling can also be accomplished by changing the think times, but this approach would be less exact because of the feedback effects, which are independent of the think times.

Other Subsystem Models

Real workloads are neither of these extremes, falling somewhere in between for reasons that will be explained in the next section. One could conceive of a sufficiently complex workload generator that would indirectly emulate the feedback behavior of a real system. Such a workload generator might be a closed subsystem model with a very large population and very complex think time distributions. Another option might be a combination of the workloads used in open and closed subsystem models, with (hopefully) less complex think time distributions. I am not aware of any successful attempts to construct such a workload generator. This dissertation proposes a more direct solution.

3.3.4 Storage Subsystem Research

This section describes many examples of storage subsystem models being used in design-stage research. The purpose of this section is to establish that, despite its flaws, the methodology described above is commonly utilized by storage subsystem designers.

Disk Request Schedulers

Disk (or drum) request schedulers have been an important system software component since the introduction of mechanical secondary storage into computer systems over 25 years ago [Denning67, Seaman66]. Over the years, many researchers have introduced, modified and evaluated disk request scheduling algorithms to reduce mechanical delays. For example, [Coff72, Gotl73, Oney75, Wilhelm76, Coffman82] all use analytic open subsystem models to compare the performance of previously introduced seek-reducing scheduling algorithms (e.g., First-Come-First-Served, Shortest-Seek-Time-First and SCAN). [Teorey72, Hofri80] use open subsystem simulation models for the same purpose. [Daniel83] introduces a continuum of seek-reducing algorithms, V-SCAN(R), and uses open subsystem simulation (as well as a real implementation tested in a user environment) to show that VSCAN(0.2) outperforms previous algorithms. [Seltzer90] and [Jacobson91] introduce algorithms that attempt to minimize total positioning times (seek plus rotation) and use closed and open subsystem simulation models, respectively, to show that they are superior to seek-reducing algorithms. [Worthington94] uses an open subsystem model to re-evaluate previous algorithms and show that they should be further modified to recognize and exploit on-board disk caches. All of this research in disk request scheduling algorithms relied upon storage subsystem models for design-stage performance comparisons.

Some previous researchers have recognized that open subsystem models can mispredict disk scheduler performance. For example, [Geist87a] compares simulation results using an open, Poisson request arrival process to measured results from a real implementation, finding that the simulator mispredicted performance by over 500 percent. Geist, et al., concluded from this that no open subsystem model provides useful information about disk scheduling algorithm performance. There are several problems with their work. Most importantly, they rated their implementation using an artificially constructed workload (of the form used in closed subsystem models) rather than real user workloads. It is not at all surprising that their open subsystem model failed to replicate the results. Also, they used unrealistic synthetic arrival times for the open subsystem model rather than traced arrival times. They went on to construct a simulation workload that better matches their artificial system workload. In a subsequent paper [Geist94], Geist and Westall exploited a pathological aspect of their artificial workload to design a scheduling algorithm that achieves anomalous improvements in disk performance.

Disk Striping

As disk performance continues to fall relative to the performance of other system components (e.g., processors and main memory), it becomes critical to utilize multiple disks in parallel. The straight-forward approach, using multiple independently-addressed drives, tends to suffer from substantial load balancing problems and does not allow multiple drives to cooperate in servicing requests for large amounts of data. Disk striping (or interleaving) spreads logically contiguous data across multiple disks by hashing on the logical address [Kim86, Salem86]. The performance impact of disk striping has been studied with both open subsystem models [Kim86, Kim91] and closed subsystem models [Salem86]. The load balancing benefits of disk striping have been demonstrated with open subsystem models [Livny87, Ganger93a]. The stripe unit size (i.e., the quantity of data mapped onto one physical disk before switching to the next) is an important design parameter that has also been studied with both open subsystem models [Livny87, Reddy89] and closed subsystem models [Chen90]. Storage subsystem models have also been used to examine other design issues in striped disk subsystems, including spindle synchronization [Kim86, Kim91] and disk/host connectivity [Ng88].

Redundant Disk Arrays

As reliability requirements and the number of disks in storage subsystems increase, it becomes important to utilize on-line redundancy. The two most popular storage redundancy mechanisms are replication (e.g., mirroring or shadowing) and parity (e.g., RAID 5). Both have been known for many years (e.g., [Ouchi78]). Storage subsystem models have been used to evaluate design issues for both replication-based redundancy (e.g., [Bitton88, Bitton89, Copeland89, Hsiao90]) and parity-based redundancy (e.g., [Muntz90, Lee91, Menon91, Holland92, Menon92, Ng92, Cao93, Hou93, Hou93a, Stodolsky93, Treiber94, Chen95]). Comparisons of replication-based and parity-based redundancy have also relied largely upon storage subsystem models (e.g., [Patterson88, Chen91, Hou93, Hou93b, Mourad93]) and measurements of prototypes under similar workloads (e.g., [Chen90a, Chervenak91]).

Dynamic Logical-to-Physical Mapping

Disk system performance can be improved in many environments by dynamically modifying the logical-to-physical mapping of data. This concept can be applied in two ways to improve performance for reads and for writes, respectively. To improve read performance, one can occasionally re-organize the data blocks to place popular blocks near the center of the disk and cluster blocks that tend to be accessed together. [Ruemmler91] uses an open subsystem model to evaluate the benefits of such an approach. [Wolf89, Vongsathorn90], on the other hand, measure this approach by implementing it in real systems. To improve write performance, one can write data blocks to convenient locations (i.e., locations that are close to the disk's read/write head) and change the mapping, rather than writing the data to the location indicated by a static mapping, which may require a significant mechanical positioning delay. Open subsystem models have been used to evaluate this approach for non-redundant storage systems (e.g., [English91]) and mirrored disk systems (e.g., [Solworth91, Orji93]). Simple equations for the disk service time improvements provided by dynamically mapping parity locations and/or data locations in a RAID 5 disk array are derived in [Menon92].

Disk Block Caches

Disk block caches are powerful tools for improving storage subsystem performance. Most design-stage studies of disk cache designs use open subsystem models, relying on traces of disk requests collected from user environments to reproduce realistic access patterns. For example, [Busch85, Smith85, Miyachi86] use trace-driven simulation to quantify the value of write-thru disk block caches and determine how they should be designed. Storage subsystem models have also been used to investigate design issues for write-back disk block caches located in the on-board disk drive control logic [Ruemmler93, Biswas93] or above the disk drive (e.g., in main memory or an intermediate controller) [Solworth90, Carson92, Reddy92]. Disk block cache design issues specific to parity-based redundant disk arrays have also been examined with open subsystem models (e.g., [Menon91, Brandwajn94, Treiber94]). Data prefetching is an extremely important aspect of disk block caching that has also been studied with open subsystem models [Ng92a, Hospodor94]. Finally, storage subsystem models have been used to evaluate the performance benefits of on-board buffers for speed-matching media and bus transfers [Mitsuishi85, Houtekamer85].

3.4 Summary

Previous work relating both to request criticality and to system-level modeling is scarce, leaving considerable room for improvement. This chapter describes this previous work and its short-comings. This chapter also describes storage subsystem models and establishes the fact that they represent a very popular tool for design-stage performance evaluation of storage subsystems. The workload generators and performance metrics commonly used with standalone storage subsystem models ignore differences in how individual I/O request response times affect system behavior, leading directly to the problems addressed in this dissertation.

CHAPTER 4

Proposed Methodology

This dissertation proposes an alternative approach to evaluating the performance of storage subsystem designs. Rather than focusing on the storage subsystem in a vacuum, the scope of the model is expanded to include all major system components, so as to incorporate the complex performance/workload feedback effects. The resulting **system-level model** is exercised with a set of application processes, and system performance metrics (e.g., the mean elapsed time for user tasks) are produced. This section describes the proposed methodology in detail. Approaches to workload generation are described with special attention given to avoiding the problems encountered with storage subsystem models.

4.1 System-Level Models

A system-level model consists of modules for each major system component and interfaces to the outside world (e.g., users and other systems).[1] Processes execute within a system-level model in a manner that imitates the behavior of the corresponding system. Also, external interrupts may arrive at the interfaces, triggering additional work for the system.

There are several options for the degree of abstraction and the granularity of the events in a system-level simulation model. For example, one can model the system at the instruction-level wherein every instruction of each involved process (and possibly the operating system) is simulated. A second alternative would be to model each memory reference, abstracting away the particulars of the instructions. Yet a third option might model system activity with higher-level system events (e.g., system calls and interrupts). More detailed models allow for more accurate results but require more implementation effort, more input (both workload and configuration) and more simulation time. The correct level at which to model system activity depends largely upon how sensitive system performance is to small changes in the components to be studied. The three options listed above are each appropriate for some studies (e.g., CPU design, memory system design and I/O subsystem design, respectively). I believe (and the validation results indicate) that the third option is sufficient for evaluating storage subsystem designs because of the large granularity of disk activity.

[1] 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.

Figure 4.1 shows an example system that includes processors, main memory, applications, operating system software, several buses, a bus adapter that also handles interrupt control activities, storage subsystem components and some other I/O devices (network and user interface). Each of these system components is very briefly described below, together with discussion of how they might be modeled in a performance evaluation tool targeted for storage subsystem designs.

Central Processing Units (CPUs)

Streams of instructions execute on the CPUs to accomplish the tasks of users. In a system-level model, a CPU can be modeled as a resource that decrements computation times between events. When the computation time reaches zero, an event "occurs," changing the state of the modeled system in some way. Generally, a CPU can only work on one task (i.e., process or interrupt service routine) at a time. That is, only one computation time can be decremented by the CPU resource during each unit of simulated time.

Main Memory

Main memory is the working space for tasks executing on the CPUs. The operating system uses the physical page frames that comprise main memory to cache virtual memory pages and file blocks. In a system-level model used for studying I/O behavior, main memory can simply be modeled as a cache for disk blocks (i.e., virtual memory pages and file blocks). The level of detail in the model will dictate whether or not all data movement into and out of main memory must be simulated.

Applications

The applications are the inputs to the computer system (together with externally generated interrupts and data), dictating the tasks performed. An application consists of one or more processes. I use the term **process** to refer to any instruction stream other than an interrupt service routine, independent of the virtual memory context. So, multiple threads that share a common context are each processes. I treat system call and exception service routines as part of the process, because they are generally executed within the context and flow of the process. A process can be modeled as a sequence of events separated by computation times. The exact form of these events depends upon the level of detail in the model and the approach to workload generation (see below).

Operating System Software

Operating system software provides commonly required functionality for applications and handles direct interactions with system hardware. Operating system functions are initiated by interrupts, exceptions and system calls made by application processes. Three important functions of most operating systems are process control, memory management and interrupt handling. Process control includes process creation/deletion, process enabling/disabling (e.g., sleep/wakeup), context switching between processes and time-sharing of CPU re-

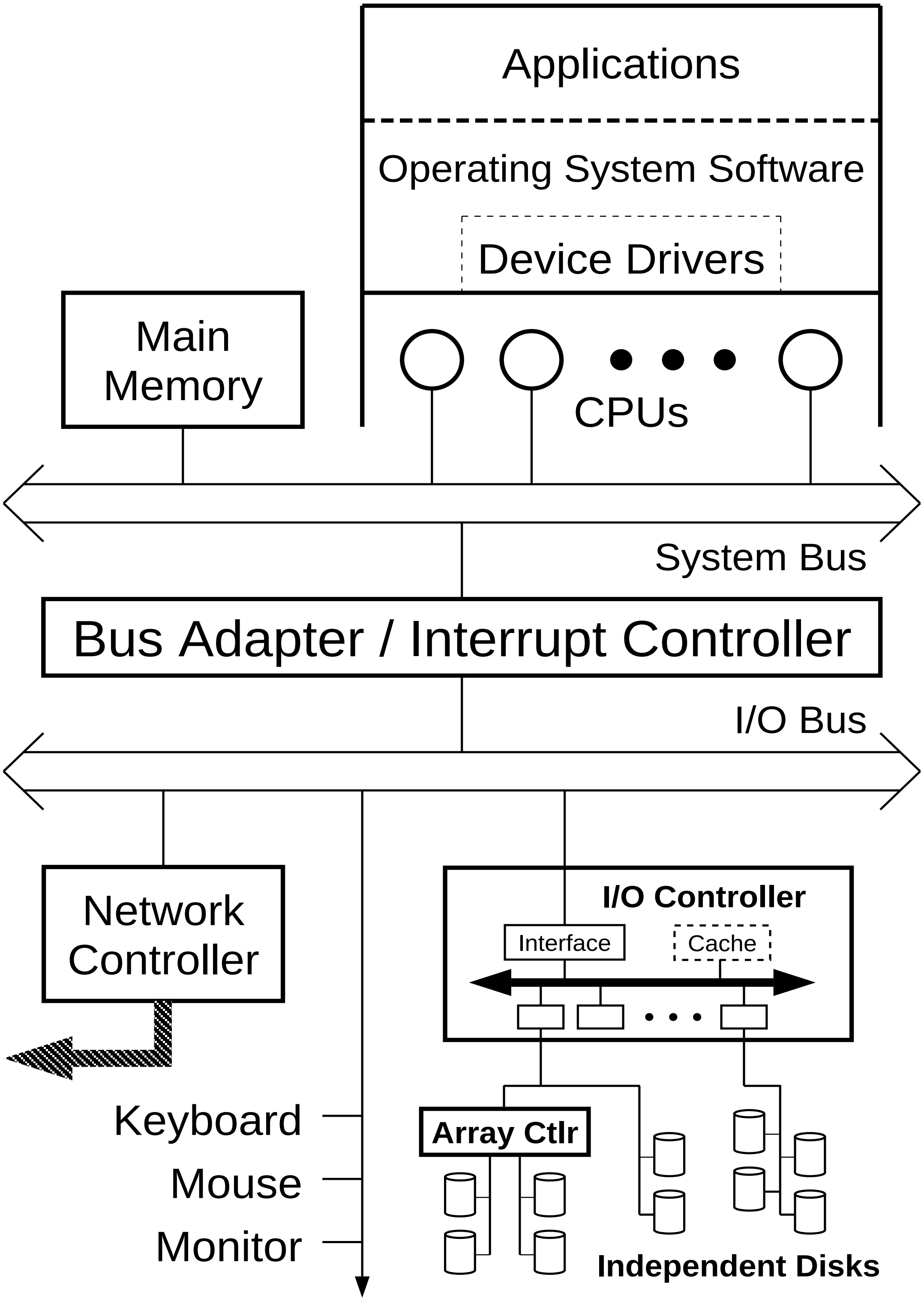


Figure 4.1: Block Diagram of a Computer System.

sources. Memory management policies involve allocation, replacement, flushing and locking decisions. An interrupt service routine executes when an interrupt arrives at the CPU, updating system software state and/or hardware controller state. Operating system software executes on CPUs and can therefore be modeled in the same manner as application software (i.e., events separated by computation times). Process control and memory management policy code can be taken almost directly from the operating system being modeled, after altering the interfaces and structures to fit into the simulation environment [Thekkath94].

Interrupt Controller

An interrupt controller tracks the pending interrupts in the system and routes them to the CPUs for service. The state of the interrupt controller is updated when new interrupts are generated and when a CPU begins handling an interrupt. The interrupt controller can be modeled as a simple resource that performs exactly the above functions. The time delays for these functions can safely be assumed to be zero in all but the most detailed models.

Non-storage I/O Devices

Non-storage I/O devices communicate with the world outside of the computer system. User-interface devices (e.g., monitors, keyboards, mice, printers, scanners, etc...) make computer systems useful to people. Network controllers allow the computer to communicate with other computer systems. There are three basic options regarding how to incorporate these devices into a system-level model: (1) ignore them and emulate a standalone computer system operating in batch mode, (2) model the workload generated by these devices in a manner analogous to the open and closed subsystem models described earlier, or (3) expand the scope of the model to incorporate these devices.

Storage Subsystem Components

The storage subsystem components perform the same functions as described earlier for the conventional methodology. The same modeling approaches for the components also apply, with the exception of the device driver. The device driver is part of the operating system and should be incorporated into the model as such, with computation times between events. However, the basic device driver functions described earlier remain unchanged.

4.2 Performance Metrics

The two most commonly used system performance metrics are elapsed times (averages, variances and/or distributions) and throughput for user tasks. The **elapsed time** for a task is the wall-clock time from when it is initiated (e.g., by a user command or a batch script) to when completion is indicated to the initiator. **Task throughput** is a measure of the number of tasks completed per unit of time. Secondary performance measures, such as false idle time, context switch count, storage I/O request count, storage performance metrics, interrupt count, main memory miss rate (for virtual memory pages and file blocks) and memory copy count, can also be used to help explain primary metric values.

4.3 Workload Generation

A system workload consists of a set of processes, a set of external interrupts, the times at which they start and their interdependencies. The external interrupts are generated by network and user interface hardware, causing an interrupt handler and possibly one or more processes to execute. As described earlier, a process can be viewed as a sequence of events separated by computation times. The particular events used will depend on the level of detail in the simulator. Each distinct event type represents a significant change to the simulated system's state. Section A.2.2 describes the process events in my simulator, but many other options exist. The critical aspect of an input workload 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 between system performance and the observed workload, causes must be part of the input workload and effects must not.

If the responsiveness of the system can affect the sequence of events in a process, this approach (which relies on pre-defined, static sequences) breaks down. For example, a server process that performs specific actions based on the messages sent by multiple independent clients may behave improperly when the order of client message arrivals changes. More work is needed to identify appropriate ways of incorporating these effects into a system-level model.

Note that my definition of a system workload does not require that the entire set of processes be present in the system initially, although at least a few generally are. Processes are created and deleted by other processes. At various points during execution, a process may also block and wait for a system event (e.g., such as a lock becoming available, an I/O request completion, a timer expiration or a key-stroke arrival) caused by an external interrupt or another process. In many environments, external activities are triggered by process activities (e.g., interactive users and remote procedure calls). The result is a complex set of interdependencies among processes and external interrupts. While some of these interactions can be obviated by assuming that certain components (e.g., the network interface or the user) are not active, others must be incorporated into a system-level model. The remainder of this section describes the various interdependencies.

Process–Process Dependencies

Processes can limit the progress of other processes in several ways. For example, since only one process can execute on a CPU at a time, one process's use of a CPU can impede the progress of another. Also, processes can compete for ownership of various other resources within the system, such as locks and memory blocks. These indirect dependencies should be included as part of the model. More direct dependencies, such as parent/child relationships and inter-process communication, must be specified in the input workload and handled properly by the model.

Inter-process communication (e.g., UNIX *pipe* and *send/recv* commands) creates producer/consumer dependencies between processes. That is, one process may only be able to proceed after receiving particular messages from another process. This can be included in a system-level model by incorporating the synchronization events into the process event sequences and emulating the corresponding dependencies.

In a UNIX system, a new process is created by an existing process via the *fork* system call. The new **child process** is an exact duplicate of the original **parent process**. The parent process usually gives up the CPU in favor of the child process initially, giving it the opportunity to begin its work. This is especially true of the *fork* variant in which the parent and child processes share the parent's virtual memory context. In this case, the system does not allow the parent process to execute again until the child process either initiates an *exec* system call or it exits. Also, many parent processes (e.g., *shell* programs) are designed to wait until child processes complete their tasks and exit.

Interrupt–Process Dependencies

As external interrupts often trigger or enable processes, the progress of a process will often depend directly upon the arrival of an external interrupt. For example, key-strokes from a user can cause one or more processes to be created and executed. As another example, network messages (e.g., remote procedure calls) may wake up a server process. This form of dependency should be incorporated directly into a system-level model. The point at which a process stops and waits is part of the process event sequence. The re-enabling of a process by an interrupt is an event in the corresponding interrupt service routine.

Process–Interrupt Dependencies

Some external interrupt arrival times depend directly upon process activity, especially output activities. For example, an interactive user will often wait to observe the results of a previous command before entering the next. Also, network traffic from other computer systems often consists of responses to previously sent messages (e.g., remote procedure calls and remote logins). The time from when a process performs an event that triggers external activity to when the corresponding interrupt arrives can be referred to as the **user think time** (or **network think time**). With these definitions, transmission delays external to the system are included in the think times. This behavior is analogous to the workload generator of a closed subsystem model and can be incorporated into a system-level model as such with the assumption that the think times do not change based on responsiveness.

Interrupt–Interrupt Dependencies

External interrupt arrival times can also depend upon previous interrupt arrivals. Most of these dependencies are indirect, via the dependencies described above, or caused by resource conflicts outside of the system (e.g., network bandwidth). However, clock-based interrupts often arrive at regular intervals. For example, an NFS client periodically checks with the server to see if cached file data have been modified [Sandberg85]. As another example, an external sensor might interrupt the system periodically with sampled data. This behavior (together with interrupt arrivals that depend on nothing, such as a user that chooses to check his/her electronic mail in the morning) is analogous to the workload generator of an open subsystem model and can be incorporated into a system-level model as such.

4.4 Summary

This chapter describes a methodology for evaluating storage subsystem designs using system-level models. Note that the contribution here is not a new model of system activity. Rather, it is the application of this model to storage subsystem design evaluation. To properly rate a subsystem design, it is necessary to understand how it interacts with user-level processes and overall system performance. This requires that feedback effects between individual request response times and user-level processes be part of the model (rather than part of the input workload). Also, interactions between processes (e.g., CPU contention) must be part of the model to obtain accurate system performance predictions.

CHAPTER 5

The Simulation Infrastructure

This chapter describes the simulation infrastructure used to validate the thesis of this dissertation. The simulator and support tools are only briefly described. More thorough descriptions can be found in appendix A. The workloads used for the experiments reported in chapters 6 and 7 are described. The system used both as a base for simulator configuration and as a source of traces is also described. Validation of the simulation infrastructure is presented.

5.1 The Simulator

The simulator is written in C and requires no special system software. General simulation support has been incorporated both to simplify the implementation of component modules and to improve simulation efficiency. The component modules are very detailed and can be configured in a wide variety of ways. The simulator can be configured to behave as either a standalone storage subsystem model or a system-level model.

The simulator contains modules for most secondary storage components of interest, including device drivers, buses, controllers, adapters and disk drives. Some of the major functions (e.g., request queueing/scheduling, disk block caching, logical data mapping) that can be present in several different components (e.g., operating system software, intermediate controllers, disk drives) have been implemented as separate modules that are linked into components as desired. The allowed interconnections are roughly independent of the components themselves except that a device driver must be at the "top" of a subsystem and storage devices (e.g., disk drives) must be at the "bottom." The separation of component definitions and their interconnections greatly reduces the effort required to develop and integrate new components, as well as the effort required to understand and modify the existing components [Satya86].

The simulator 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. Very briefly, the system-level model operates as follows. Processes and interrupt service routines are modeled as sequences of events (i.e., important system software state changes) 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 simulator accepts three forms of input: storage I/O request traces, system-level traces and synthetic I/O workload descriptions. The form of input determines the type of simulation used. Storage I/O request traces drive storage subsystem simulations. System-level traces drive system-level simulations. Synthetic I/O workload descriptions utilize components of the system-level to drive storage subsystem simulations.

5.2 Current Library of System-Level Traces

Several system-level traces have been captured for the work described in this dissertation. All of the traces were captured on the experimental system described below using the operating system instrumentation described in appendix A. The traced workloads fall into two categories: independent task workloads and SynRGen workloads.

Independent Task Workloads

The independent task workloads consist of a single, though often substantial, user task. Each such workload executes as a single *csh* script with the following form:

START TRACE
DO TASK
sleep 65
sync
sleep 65
STOP TRACE

I/O requests for all dirty file cache blocks generated during task execution are initiated by the syncer daemon (see section 5.3) before the first *sleep* command completes. The *sync* command and the second *sleep* command are unnecessary precautions maintained for historical reasons. Each script command is executed by a new process created by the *csh* environment. I refer to the process that performs the "DO TASK" command as the **task-executing process**. Several instances of each task have been traced, allowing generation of statistically significant results. Table 5.1 lists basic characteristics of the four independent task workloads. They are:

- *compress*: This task uses the UNIX *compress* utility to reduce a 30.7 MB file to 10.7 MB. The file being compressed is not resident in the main memory file block cache when the task begins.

- *uncompress*: This task uses the UNIX *uncompress* utility to return a 10.7 MB file to its original 30.7 MB size. The file being compressed is not resident in the main memory file block cache when the task begins.

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.	10844	53.3 ms
uncompress	144 sec.	91.0 sec.	47.1 sec.	10983	1076 ms
copytree	89.5 sec.	17.6 sec.	69.7 sec	8995	147 ms
removetree	18.2 sec.	3.05 sec.	14.9 sec	1176	15.6 ms

Table 5.1: Basic Characteristics of the Independent Task Workloads.

- *copytree*: This task uses the UNIX *cp -r* command to copy a user directory tree containing 535 files totaling 14.3 MB of storage. The source directory tree is not resident in the main memory file block cache when the task begins.

- *removetree*: This task uses the UNIX *rm -r* command to remove a directory tree containing 535 files totaling 14.3 MB of storage. The source directory tree is mostly resident in the main memory file block cache when the task begins, since the experiment removes the new copy resulting from an immediately prior *copytree* execution.

SynRGen Workloads

SynRGen is a synthetic file reference generator that operates at the system call level [Ebling94]. For trace collection, SynRGen was configured to imitate programmers performing edit/debug activity on large software systems. [Ebling94] showed that this configuration closely matches measurements of such user activity. Each programmer is emulated by a single process that executes tasks interspersed with user think times. Each task consists of a series of file accesses and possibly some computation time (e.g., to emulate compilation or program execution). Both computation times and user think times are modeled using the UNIX *sleep* command.[1] The computation times are configured to match measurements from older machines (DEC DS-5000/200s) and probably overestimate current CPU times. I refer to the computation time portion of a task's elapsed time as the **task sleep time**, since *sleep* is used to realize it. To measure system activity over a range of workloads, I have collected traces with different numbers of emulated users (1, 2, 4, 8, 16). I refer to each workload as *synrgen#*, where # is the number of users. Table 5.2 lists basic characteristics of the SynRGen workloads.

[1] For the user think times, *sleep* is not inappropriate. For computation times, however, it would be better to have the CPU utilized in some way.

Independent Task	Avg. Task Elapsed Time	Avg. Task Sleep Time	Avg. Task I/O Wait	# of I/O Requests	Avg. I/O Resp. Time
synrgen1	2.37 sec.	2.02 sec.	79 ms	3022	96.4 ms
synrgen2	3.02 sec.	2.60 sec.	97 ms	3668	86.6 ms
synrgen4	3.19 sec.	2.74 sec.	137 ms	4987	214 ms
synrgen8	3.25 sec.	2.71 sec.	232 ms	5565	232 ms
synrgen16	3.26 sec.	2.55 sec.	679 ms	5598	679 ms

Table 5.2: Basic Characteristics of the SynRGen Workloads.

5.3 The Experimental System

The base system for my experiments (both simulation and implementation) is an NCR 3433, a 33 MHz Intel 80486 machine equipped with 48 MB of main memory.[2] Most of the experiments use an HP C2247 disk drive, which is a high performance, 3.5-inch, 1 GB SCSI storage device [HP92]. Table 5.3 lists some basic performance characteristics of this disk drive and [Worthington94a] provides a thorough breakdown of simulator configuration parameter values. The operating system is UNIX SVR4 MP, AT&T/GIS's production operating system for symmetric multiprocessing. The default file system, *ufs*, is based on the Berkeley fast file system [McKusick84]. The virtual memory system is similar to that of SunOS [Gingell87, Moran87] and file system caching is well integrated with the virtual memory system. Unless otherwise noted, the scheduling algorithm used by the device driver is LBN-based C-LOOK, which always schedules the request with the smallest starting address that exceeds the most recent starting address (or zero, if no pending requests are beyond the most recent request). Command queueing at the disk is disabled. 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 wakes up periodically and writes out dirty buffer cache blocks. The syncer daemon in UNIX SVR4 MP operates differently than the conventional "30 second sync." It awakens once each second and sweeps through a fraction 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 [Carson92, Mogul94]) but does not completely alleviate the phenomenon.

In order to accurately model the experimental system, an extensive set of parameters was obtained from published documentation, operating system source code and direct observation of system-level activity (as captured by the system-level trace instrumentation) and SCSI bus activity (as captured by a logic analyzer connected to the SCSI bus).

[2] For most of the experiments, including trace collection, the available physical memory is partitioned into 40 MB for normal system use and 8 MB for the trace buffer (see section A.2.2).

HP C2247 Disk Drive	
Formatted Capacity	1.05 GB
Rotation Speed	5400 RPM
Data Surfaces	13
Cylinders	2051
512-Byte Sectors	2054864
Zones	8
Sectors/Track	56-96
Interface	SCSI-2
256 KB Cache, 2 Segments	
Track Sparing/Reallocation	

Table 5.3: Default Characteristics of the HP C2247 Disk Drive.

5.4 Validation

This section validates the simulation infrastructure described above by comparing its performance predictions to performance measurements of a real system. The system used for this validation is described above. Various performance metrics predicted by the simulator and measured on the real system are compared. The storage subsystem components are more thoroughly validated by comparing request response time distributions. An additional level of validation is provided by modifying the simulator and the real system and comparing the predicted and measured performance change.

5.4.1 Predicted vs. Measured Performance

To verify that the simulator correctly emulates the experimental system described above, I collected several performance measurements in the form of system-level traces (see A.2.2). 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. Tables 5.4–5.7 show the results of these comparisons for four different independent task workloads. Most of the simulator values are within 1% of the corresponding measured value. The largest difference observed (among these and other validation experiments) was 5%.

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 5.4: Simulator Validation for the *compress* Workload.

Metric	Measured	Simulated	% Diff.
Elapsed Time	144 sec	143 sec	-0.7%
CPU utilization	62.4 %	63.3 %	1.4%
Number of interrupts	79513	78452	-1.3%
Number of I/O requests	10983	10993	0.1%
Number of I/O waits	166	163	-1.8%
Average I/O wait time	284 ms	286 ms	0.7%
Average I/O access time	8.32 ms	8.36 ms	0.5%
Average I/O response time	1076 ms	1067 ms	-0.8%

Table 5.5: Simulator Validation for the *uncompress* Workload.

Metric	Measured	Simulated	% Diff.
Elapsed Time	89.5 sec	88.7 sec	-0.9%
CPU utilization	19.3 %	19.5 %	1.0%
Number of interrupts	62872	62474	-0.6%
Number of I/O requests	8995	8995	0.0%
Number of I/O waits	4584	4531	-1.2%
Average I/O wait time	15.2 ms	15.1 ms	-0.7%
Average I/O access time	11.1 ms	10.9 ms	-1.8%
Average I/O response time	147 ms	145 ms	-1.4%

Table 5.6: Simulator Validation for the *copytree* Workload.

Metric	Measured	Simulated	% Diff.
Elapsed Time	18.2 sec	18.4 sec	1.1%
CPU utilization	15.1 %	15.6 %	3.2%
Number of interrupts	21756	21686	-0.3%
Number of I/O requests	1176	1170	-0.1%
Number of I/O waits	1103	1103	0.0%
Average I/O wait time	13.5 ms	13.7 ms	1.5%
Average I/O access time	13.9 ms	14.0 ms	0.7%
Average I/O response time	15.6 ms	15.6 ms	-0.1%

Table 5.7: Simulator Validation for the *removetree* Workload.

5.4.2 Predicted vs. Measured Response Time Distributions

Greater insight into the validity of a storage subsystem model can be gained by comparing measured and simulated response time distributions [Ruemmler94]. The storage subsystem simulator has therefore been validated by exercising various disk drives and capturing traces of the resulting I/O activity. Using the observed inter-request delays, each traced request stream was also run through the simulator, which was configured to emulate the corresponding real subsystem. For each disk, this process was repeated for several synthetic workloads with varying read/write ratios, arrival rates, request sizes, and degrees of sequentiality and locality. The measured and simulated response time averages match to within 0.8% for all validation runs. Figures 5.1–5.5 show distributions of measured and simulated response times for a sample validation workload of 10,000 requests. As with most of our validation results, one can barely see that two curves are present. [Ruemmler94] defines the root mean square horizontal distance between the two distribution curves as a **demerit figure** for disk model calibration. The demerit figure for each of the curves shown is given in the corresponding caption. The worst-case demerit figure observed over all validation runs was only 2.0% of the corresponding average response time.

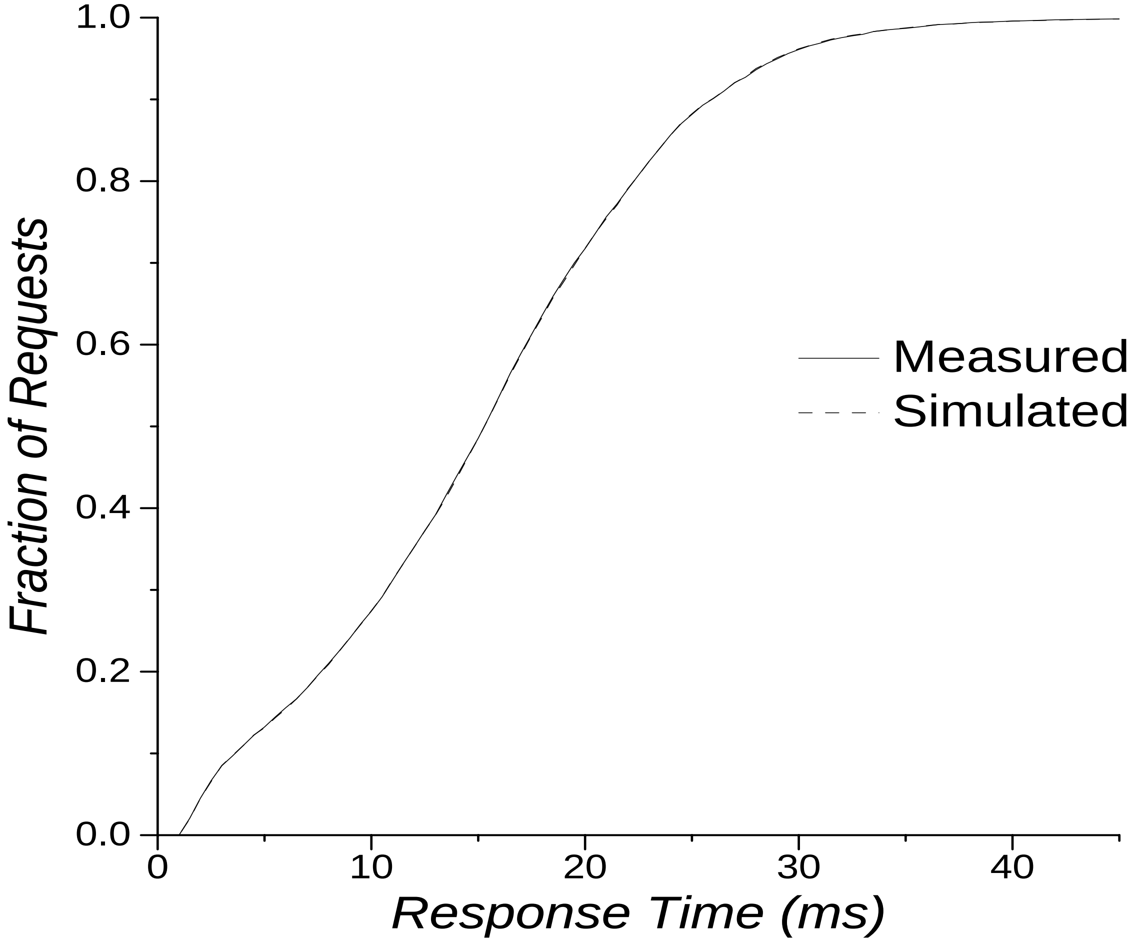


Figure 5.1: Measured and Simulated Response Time Distributions for an HP C2247A Disk Drive. The demerit figure for this validation run is 0.07 ms, or 0.5% of the corresponding mean response time. Characteristics of the HP C2247A can be found in table 5.3 and in [HP92, Worthington94]. The validation workload parameters are 50% reads, 30% sequential, 30% local [normal with 10000 sector variance], 8KB mean request size [exponential], interarrival time [uniform 0–22 ms].

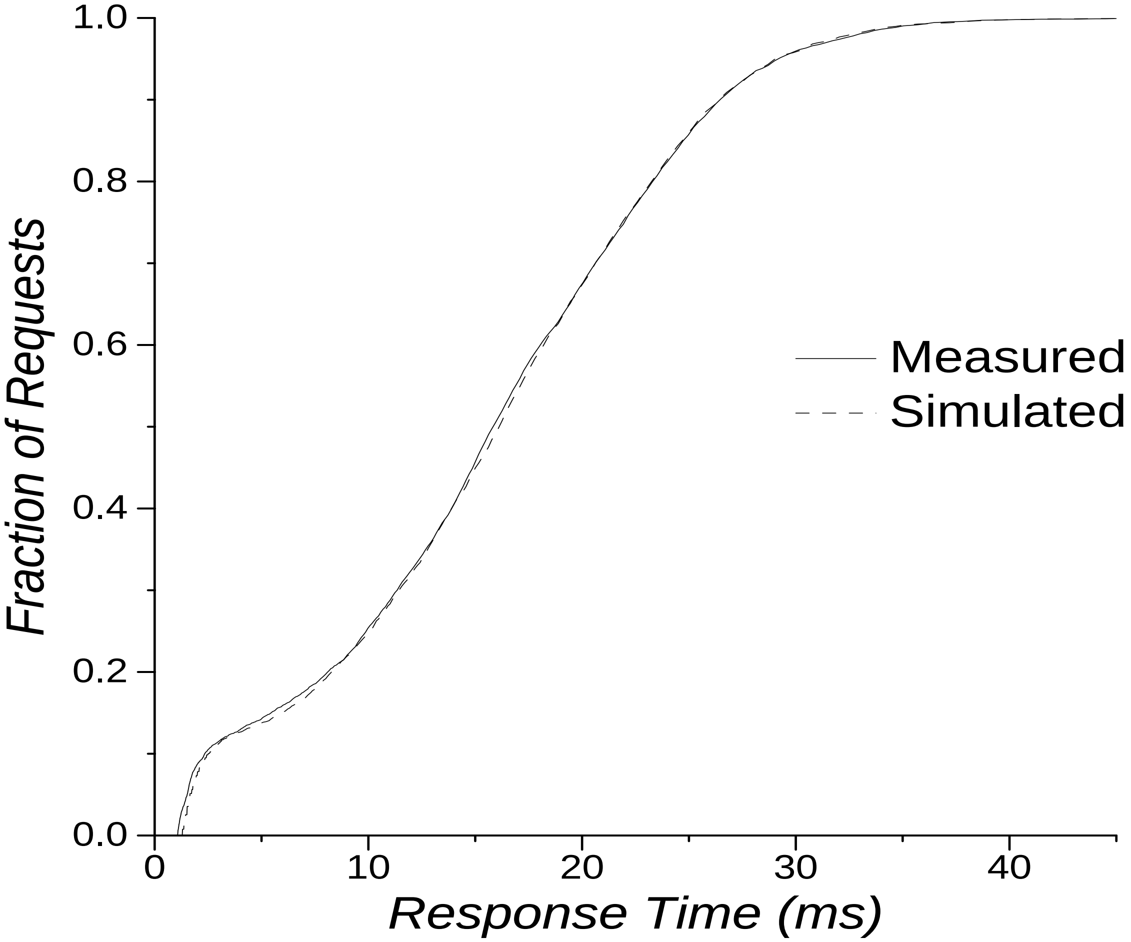


Figure 5.2: Measured and Simulated Response Time Distributions for a DEC RZ26 Disk Drive. The demerit figure for this validation run is 0.19 ms, or 1.2% of the corresponding mean response time. The validation workload parameters are 50% reads, 30% sequential, 30% local [normal with 10000 sector variance], 8KB mean request size [exponential], interarrival time [uniform 0–22 ms].

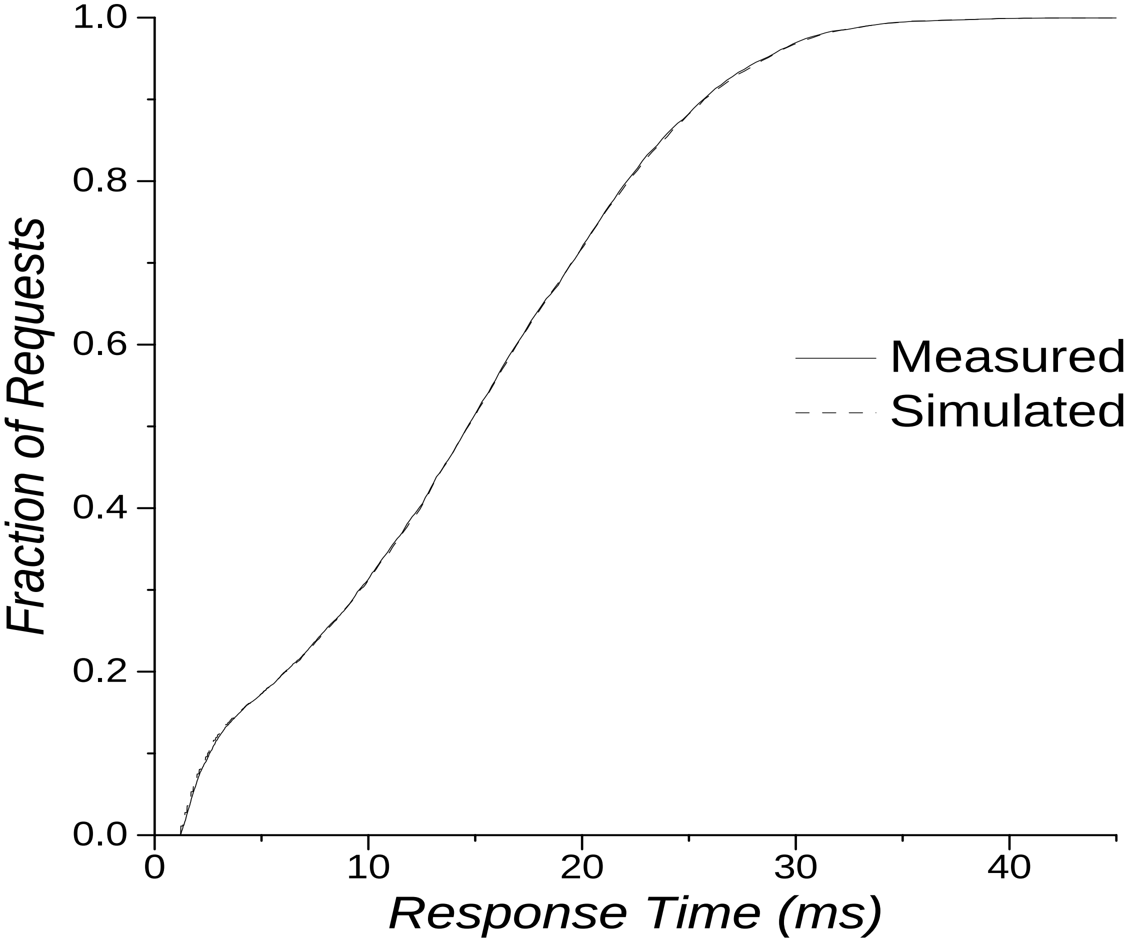


Figure 5.3: Measured and Simulated Response Time Distributions for a Seagate Elite ST41601N Disk Drive. The demerit figure for this validation run is 0.075 ms, or 0.5% of the corresponding mean response time. Characteristics of the Seagate ST41601N can be found in [Seagate92, Seagate92a]. The validation workload parameters are 50% reads, 30% sequential, 30% local [normal with 10000 sector variance], 8KB mean request size [exponential], interarrival time [uniform 0–22 ms].

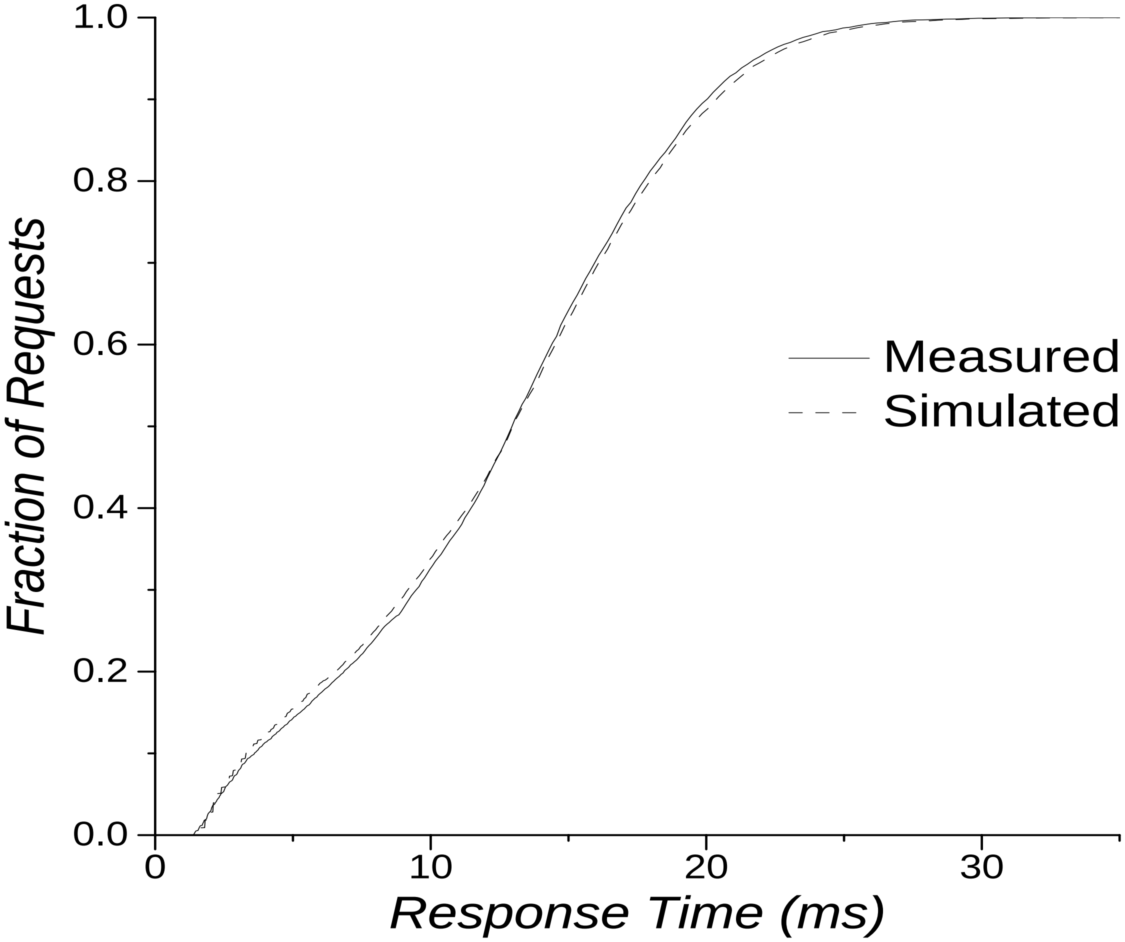


Figure 5.4: Measured and Simulated Response Time Distributions for an HP C2490A Disk Drive. The demerit figure for this validation run is 0.26 ms, or 2.0% of the corresponding mean response time. Characteristics of the HP C2490A can be found in [HP93]. The validation workload parameters are 50% reads, 30% sequential, 30% local [normal with 10000 sector variance], 8KB mean request size [exponential], interarrival time [uniform 0–22 ms].

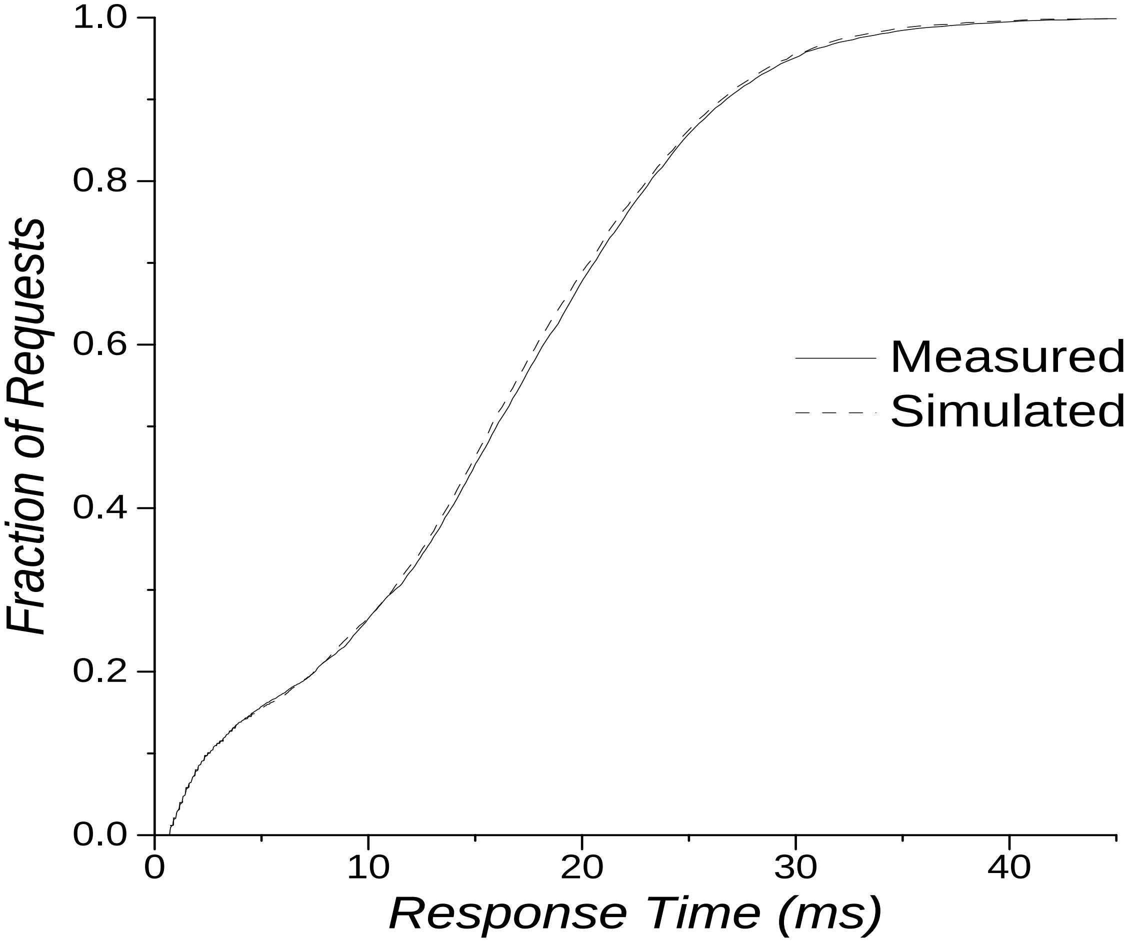


Figure 5.5: Measured and Simulated Response Time Distributions for an HP C3323A Disk Drive. The demerit figure for this validation run is 0.31 ms, or 1.9% of the corresponding mean response time. Characteristics of the HP C3323A can be found in [HP94]. The validation workload parameters are 50% reads, 30% sequential, 30% local [normal with 10000 sector variance], 8KB mean request size [exponential], interarrival time [uniform 0–22 ms].

Metric	Measured Improvement	Simulated Improvement
Elapsed Time	3.0 %	3.0 %
Average I/O response time	9.2 %	9.4 %
Average I/O access time	4.2 %	4.3 %

Table 5.8: Measured and Simulated Performance Improvement for the *compress* Workload. The improvements come from using the C-LOOK disk scheduling algorithm rather than First-Come-First-Served.

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 %

Table 5.9: Measured and Simulated Performance Improvement for the *uncompress* Workload. The improvements come from using the C-LOOK disk scheduling algorithm rather than First-Come-First-Served.

5.4.3 Predicted vs. Measured Performance Improvements

Once a simulator's ability to emulate the corresponding 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 the measurements to the change in performance predicted with the simulator. Of course, the possible modifications are limited to those that can be made on the real system. So, for example, one can not verify the simulator's ability to predict how storage components that do not yet exist will affect performance.

To validate that the simulator correctly predicts performance changes, I experimented with two disk scheduling algorithms in the device driver. Tables 5.8–5.11 compare 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 percentages in each table are averages across five independent traces of the corresponding benchmark. The simulator's predictions match the measured values very closely. The largest difference was observed with the recursive copy experiment (table 5.10). I believe that variations in file placement account for the majority of the error.

Metric	Measured Improvement	Simulated Improvement
Elapsed Time	6.1 %	6.5 %
Average I/O response time	38.2 %	40.8 %
Average I/O access time	6.8 %	7.2 %

Table 5.10: Measured and Simulated Performance Improvement for the *copytree* Workload. The improvements come from using the C-LOOK disk scheduling algorithm rather than First-Come-First-Served.

Metric	Measured Improvement	Simulated Improvement
Elapsed Time	-0.2 %	0.0 %
Average I/O response time	0.3 %	0.1 %
Average I/O access time	0.2 %	0.1 %

Table 5.11: Measured and Simulated Performance Improvement for the *removetree* Workload. The improvements come from using the C-LOOK disk scheduling algorithm rather than First-Come-First-Served.

5.5 Summary

This chapter outlines the simulation infrastructure used to validate the thesis of this dissertation. The benchmark workloads used for the experiments in chapters 6 and 7 are described. The simulator has been validated, resulting in a high degree of confidence in its performance predictions. The simulator can be used either as a standalone storage subsystem model or as a system-level model. As a system-level model, it represents an existence proof that accurate and efficient system-level models can be constructed.

CHAPTER 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, the lack of proper feedback can ruin the representativeness of the workload, leading to incorrect conclusions regarding performance. This chapter demonstrates this problem and the resulting effects via several examples where inaccurate feedback effects cause the conventional methodology to produce erroneous results. Using a system-level simulation model and similar storage subsystem simulation models, examples of quantitative and qualitative errors are shown. Where possible, performance values measured from the experimental system are used to corroborate the system-level model's predictions.

6.1 Storage Subsystem Workload Generators

Commonly used workload generators for storage subsystem models fall into two groups. Open subsystem models use predetermined arrival times for requests, independent of the storage subsystem's performance. Closed subsystem models maintain a constant population of requests, generating a new request whenever a previous request completes. This section describes the workload generation processes used to represent open and closed subsystem simulation models in this chapter.

6.1.1 Open Subsystem Models

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, the workload intensity does not increase (decrease) in response to improvement (degradation) in storage performance. This fact often leads to over-estimation of performance improvements, because performance changes tend to be multiplicative in the presence of request queueing. The lack of feedback also allows requests to be outstanding concurrently that would never in reality be outstanding at the same time.

To exercise open subsystem models in this chapter, I use the trace of I/O requests extracted from a system-level trace.[1] The arrival times in the trace, as well as the physical access characteristics, are maintained to recreate the observed storage I/O workload. If the storage subsystem model exactly replicates the observed storage subsystem behavior, the resulting storage performance metrics will be identical, as the feedback effects (or lack thereof) will not come into play.[2] Trace scaling, accomplished by stretching or compressing the inter-arrival times, is used when necessary to increase or decrease the workload intensity.

6.1.2 Closed Subsystem Models

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 (interspersed periods of intense activity and no activity). As a result, closed subsystem models generally under-estimate performance improvements, which often consist largely of reduced queueing delays. Also, the lack of both intense bursts of activity and idle periods can prevent the identification of optimizations related to each.

To exercise closed subsystem models in this chapter, I use the trace of I/O requests extracted from a system-level trace. The 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 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. Workload scaling, when desired, is accomplished by simply increasing or decreasing the value of N.

6.2 Quantitative Errors

By misrepresenting performance/workload feedback effects, storage subsystem models frequently over-estimate or under-estimate performance changes. This section provides several examples of this behavior.

6.2.1 Disk Request Scheduling Algorithms

Many disk request scheduling algorithms have been proposed and studied over the past 30 years (see section 3.3.4) to reduce the mechanical delays associated with disk drives. In

[1] The results are usually compared to those produced by the system-level simulator driven by the full system-level traces.

[2] For this reason, I have used open subsystem simulation driven by traces of observed disk activity in my previous work [Ganger93a, Worthington94]. The results match reality in at least one instance. There is no corresponding trace-based workload generator for closed subsystem models, which never match the reality of most workloads.

this subsection, I compare two of them. 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. The former algorithm performs no storage subsystem performance optimizations. The latter is used in many existing systems and has been shown to outperform other seek-reducing algorithms for many real workloads [Worthington94].

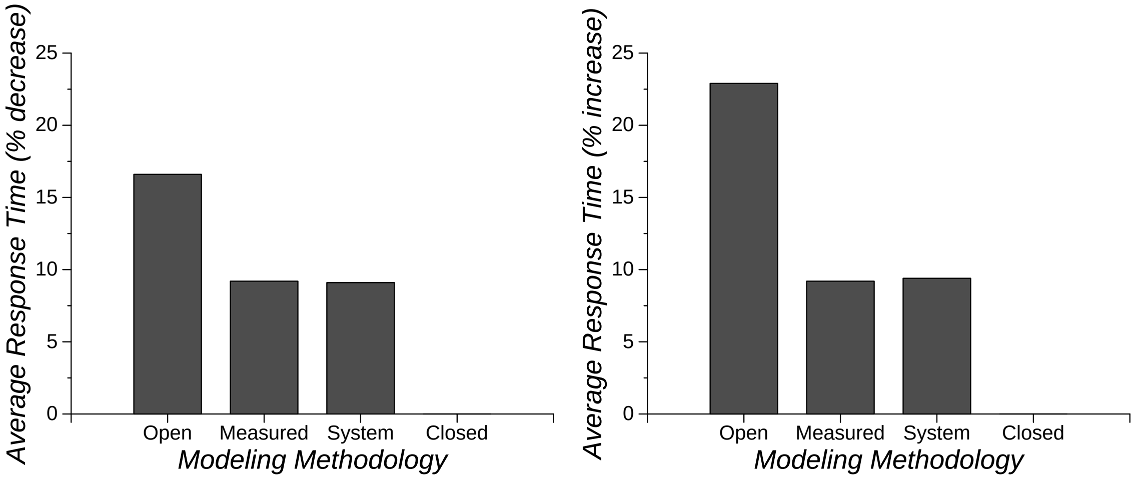
Figures 6.1–6.4 show the measured and predicted performance effects of replacing one algorithm with the other for the individual task workloads. Each figure contains two graphs, representing the change from FCFS to C-LOOK and the reverse, respectively. The distinction relates to which algorithm was used 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 performance difference measured on the experimental system, which matches the system-level model's predictions closely in every case.

In every comparison, the open subsystem model over-estimates the actual performance change and the closed subsystem model under-estimates it. This behavior matches the expectations outlined above and is consistent across all of my modeling methodology comparisons.

Two other insights can be gained from these data. First, the quantitative error associated with open subsystem modeling is much larger when the modeled subsystem services requests less quickly than the traced subsystem. This can be seen by comparing the data shown in the second graph of each figure to that shown in the first, because C-LOOK consistently outperforms FCFS. The second insight relates to the lack of improvement predicted by the closed subsystem model for the *compress* and *removetree* workloads (figures 6.1 and 6.4). The nature of the closed subsystem model is such that the scheduler never has the opportunity to make a decision, and therefore does not affect performance, unless the request population is greater than two.

6.2.2 Prediction Error for Open Subsystem Models

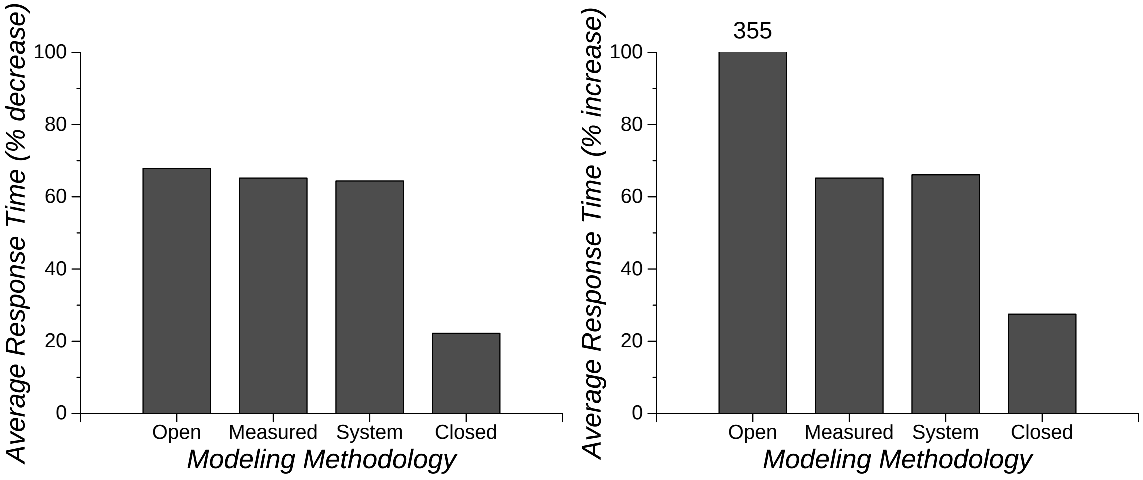
In most cases, open subsystem models approximate real system behavior better than closed subsystem models, but not as well as system-level models. This subsection quantifies and explains the performance prediction error in open subsystem models. Figure 6.5 displays the error in mean response time predictions generated by the open subsystem model (relative to the system-level model) when storage component performance changes. The speedup factors on the X-axis are applied to all storage performance attributes (e.g., bus transfer rates, disk rotation speeds and command processing overheads), so all request service times should change by roughly the same factor. Queueing delays, however, will vary depending on the burstiness (and feedback) of the request arrival pattern. FCFS disk scheduling is used, so request service order will be changed only by feedback effects in the system-level model.



(a) FCFS ⟹ C-LOOK

(b) C-LOOK ⟹ FCFS

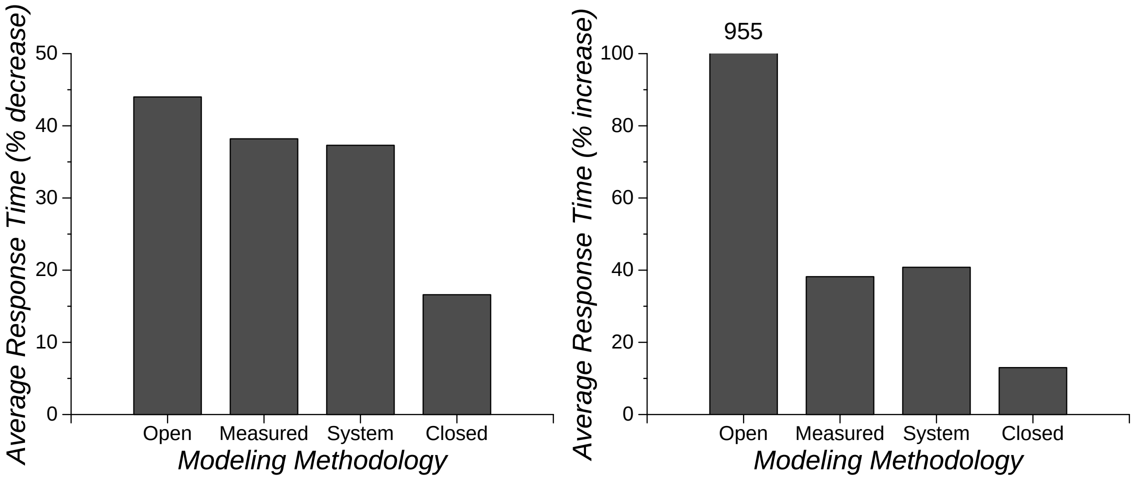
Figure 6.1: Scheduling Algorithm Comparison for the *compress* Workload. The closed subsystem model maintains a request population of 2 for both (a) and (b).



(a) FCFS ⟹ C-LOOK

(b) C-LOOK ⟹ FCFS

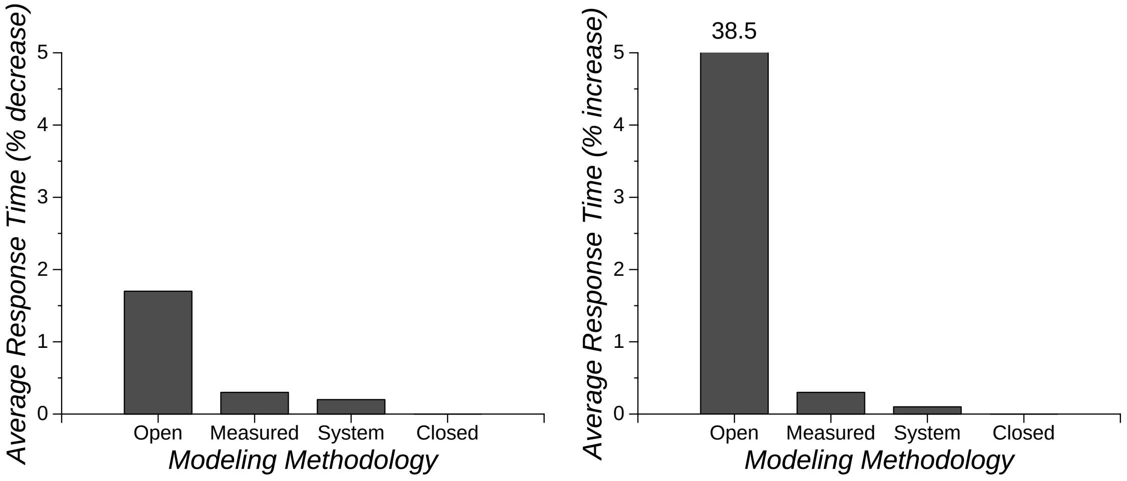
Figure 6.2: Scheduling Algorithm Comparison for the *uncompress* Workload. The closed subsystem model maintains a request population of 100 for (a) and 42 for (b), corresponding to the average populations observed with FCFS and C-LOOK. In graph (b), the open subsystem model predicts a 355% increase in the average response time.



(a) FCFS $\Longrightarrow$ C-LOOK

(b) C-LOOK $\Longrightarrow$ FCFS

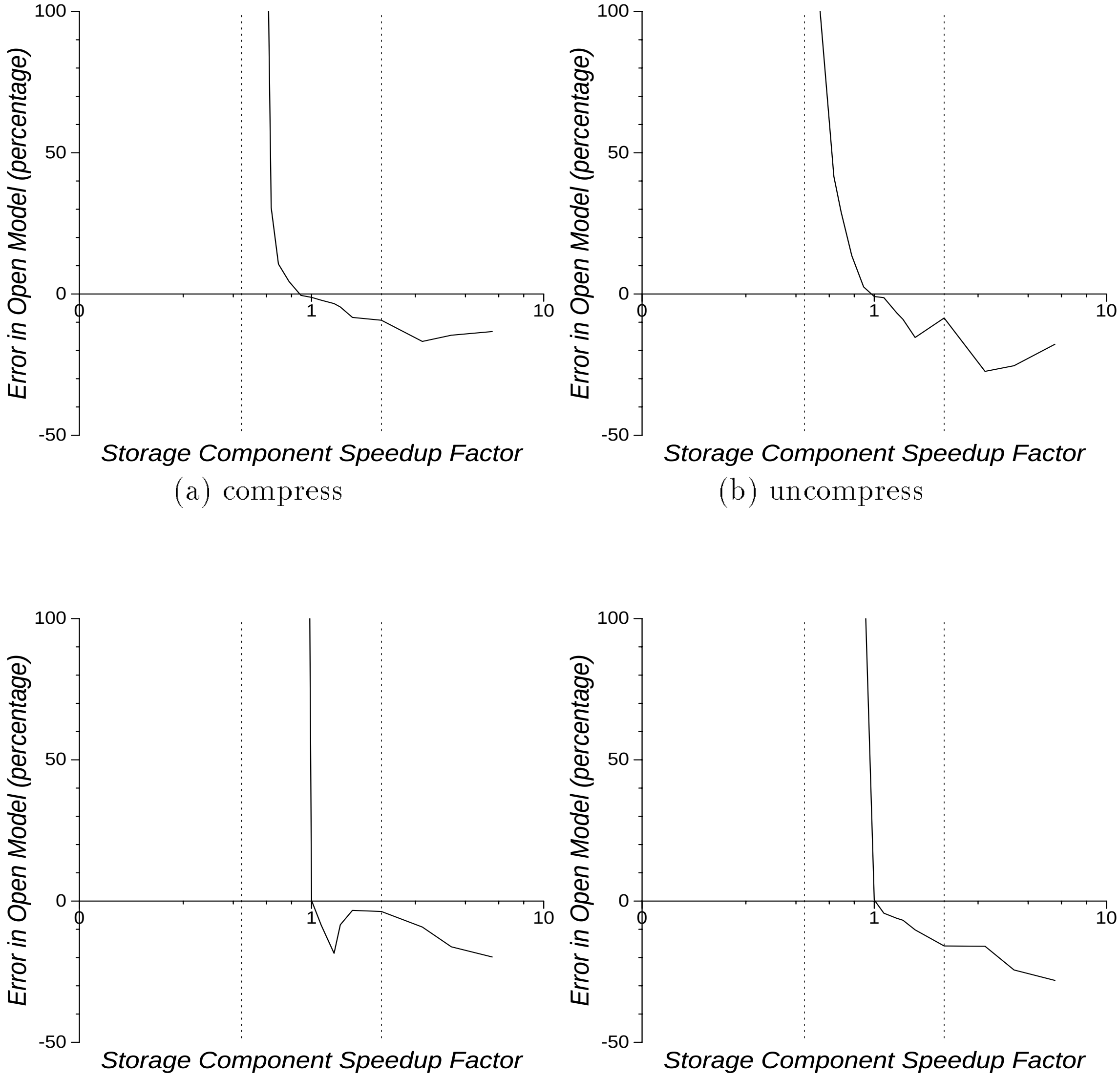
Figure 6.3: Scheduling Algorithm Comparison for the *copytree* Workload. The closed subsystem model maintains a request population of 9 for (a) and 5 for (b), corresponding to the average populations observed with FCFS and C-LOOK. In graph (b), the open subsystem model predicts a 955% increase in the average response time.



(a) FCFS $\Longrightarrow$ C-LOOK

(b) C-LOOK $\Longrightarrow$ FCFS

Figure 6.4: Scheduling Algorithm Comparison for the *removetree* Workload. The closed subsystem model maintains a request population of 1 for both (a) and (b). In graph (b), the open subsystem model predicts a 38.5% increase in the average response time.



(a) compress

(b) uncompress

(c) copytree

(d) removetree

Figure 6.5: Error in Open Subsystem Model Performance Predictions. All storage subsystem component delays are reduced by the speedup factor on the X-axis. At 1.0, simulated storage performance matches the traced subsystem. The solid line shows the error in the mean response time predicted by the open subsystem model relative to the system-level model. The two dotted lines highlight the points where storage component performance has been halved and doubled.

The general trend is that open subsystem model predictions decrease relative to system-level model predictions as performance increases.[3] For this reason, open subsystem models tend to overpredict performance changes (in either direction). The magnitude of the error grows quickly as simulated component performance decreases relative to the traced components. When the simulated components outperform the traced components, the error is much smaller. For example, the error ranges from 4% to 16% for a component speedup factor of two.

Two factors have the largest impact on the magnitude of open subsystem modeling error: request criticality and workload intensity. Workloads with many time-critical requests, relative to the number of time-noncritical requests, tend to cause problems for open subsystem models (which do not incorporate the corresponding feedback effects). For example, the error for the *removetree* workload is roughly twice as large as that for the *copytree* workload, because the former consists largely of time-critical requests. This difference is evident despite the fact that *copytree* is a much heavier workload. Heavy workloads tend to result in larger errors because request queueing emphasizes any differences. For example, the error for the *uncompress* workload is larger than that for the *compress* and *copytree* workloads, which contain more critical requests, because it demands more from the storage subsystem.

Note that the error values shown in figure 6.5 represent roughly uniform service time changes for all requests. When the service times for different requests change in different ways, the expected error becomes more difficult to predict. In general, changes to response times for time-critical and time-limited requests will cause significant error and changes to response times for time-noncritical requests will not. For example, evaluating criticality-based disk scheduling algorithms (see chapter 7) with open subsystem models results in large prediction errors in mean response time.

6.3 Qualitative Errors

While quantitative errors in performance predictions 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. This section provides several examples where this is not the case and the conventional methodology produces incorrect qualitative conclusions. The first two examples exploit, in an obvious way, the short-comings in open and closed subsystem models, respectively. Additional (less obvious) examples indicate that these short-comings can result in erroneous answers because of complex interactions and secondary performance effects.

6.3.1 Disk Request Collapsing

As described earlier, a major problem with open subsystem models is that they allow requests to be outstanding concurrently (in the model) that would never in reality be outstanding at the same time. This subsection exploits this short-coming by examining the use of request collapsing in the disk scheduler. In most systems, there are no implied ordering

[3] The error lines are not perfectly smooth for all benchmarks (e.g., *copytree*) because increasing some storage component performance characteristics (e.g., rotation speed) can affect each request's service time in different ways.

semantics among requests issued into the storage subsystem. It is this fact that allows disk schedulers the freedom to reorder requests. If two outstanding requests access the same set of disk blocks, the scheduler can select one of them for service, perform some memory-to-memory copies (if necessary) and consider both requests complete, with no externally visible side effects. For example, two write requests for the same disk block can be collapsed into a single disk access. One of them is serviced and then the two are reported complete in reverse order. Two read requests for the same block can also be collapsed into a single disk access by copying the results of one to the destination of the other. A read request and a write request for the same block can be collapsed by servicing the write, copying the source memory for the write to the destination of the read, reporting the write completion and then reporting the read completion. The benefit of disk request collapsing, of course, depends mainly upon the frequency with which overlapping requests are outstanding concurrently.

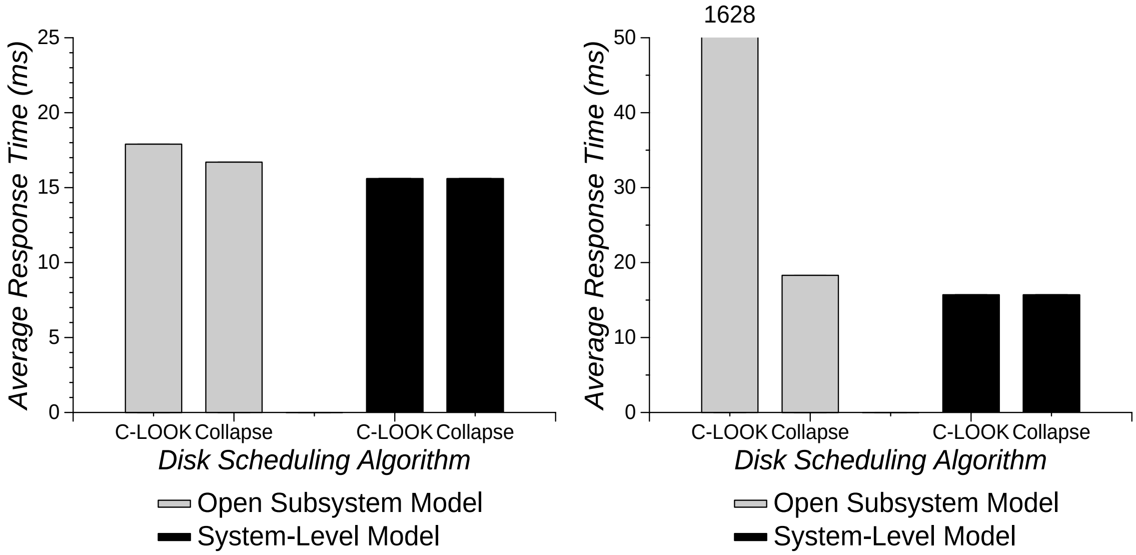
Figures 6.6 and 6.7 evaluate the addition of disk request collapsing to the C-LOOK scheduling algorithm for two different independent task workloads. Each figure contains two graphs, comparing performance with scale factors of 1 and 2. Recall that the open subsystem model scales a workload upward by shrinking the measured request inter-arrival times. The system-level model scales a workload by increasing the speed of the host system components (CPU speed and memory bandwidth). Each graph shows two pairs of bars, comparing the average response times predicted with the open subsystem model and the system-level model, respectively.

In figure 6.6, the open subsystem model predicts a reduction in the average response time of 7% with a trace scaling factor of one and of 99% with a scaling factor of two. The second workload exhibits less temporal locality and the open subsystem model predicts no improvement with a scaling factor of one. However, with a scaling factor of two, the open subsystem model predicts an average response time reduction of 45%. A naive storage subsystem designer might observe these results and be convinced that request collapsing is crucial to achieving high performance and that its importance increases as workloads become heavier.

On the other hand, the system-level model predicts that request collapsing has no effect on performance, since overlapping requests are never outstanding together in the experimental system. Like most systems, it uses main memory as a cache for disk blocks, in the form of file blocks and virtual memory pages. Concurrent requests for the same data will meet at this cache and either be combined by higher level system components or be issued to the storage subsystem in sequence. The clear result is that request collapsing in the disk request scheduler is not useful. This example highlights a major problem with ignoring performance/workload feedback effects, showing that it can lead to incorrect conclusions.

6.3.2 Flush Policies for Write-Back Disk Block Caches

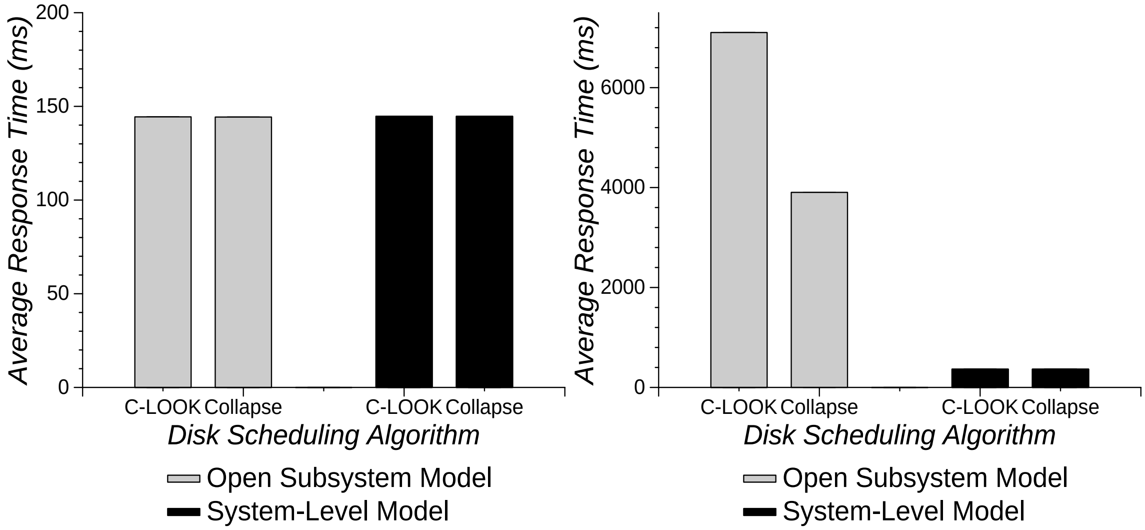
As described earlier, the major problem with closed subsystem models is that they do not incorporate any of the burstiness that characterizes real workloads, resulting in a complete lack of idle time in the storage subsystem. This subsection exploits this short-coming by comparing two flush policies for a 1 MB write-back disk cache residing on an intermediate SCSI bus controller. The first policy cleans dirty blocks only when they must be reclaimed. As a result, flush activity always impedes the progress of outstanding requests. The second



(a) Scale Factor = 1.0

(b) Scale Factor = 2.0

Figure 6.6: Disk Request Collapsing for the *removetree* workload. In (b), the open subsystem model predicts an average response time of 1628 ms for the conventional C-LOOK algorithm.



(a) Scale Factor = 1.0

(b) Scale Factor = 2.0

Figure 6.7: Disk Request Collapsing for the *copytree* workload.

policy detects idle time and flushes dirty blocks in the background. The idle detector initiates cache cleaning activity when 0.5 seconds pass with no outstanding requests [Golding95]. Cleaning activity halts whenever a new request arrives. If a dirty block must be reclaimed, it is flushed immediately, as with the base policy. The benefit of detecting and using idle time in this way depends primarily on the amount of idle time available.

Figure 6.8 compares the two flush policies using SynRGen workloads with different numbers of users. The four graphs represent 2 users, 4 users, 8 users and 16 users, respectively. Each graph shows two pairs of bars, comparing the response times predicted with the closed subsystem model and the system-level model, respectively.

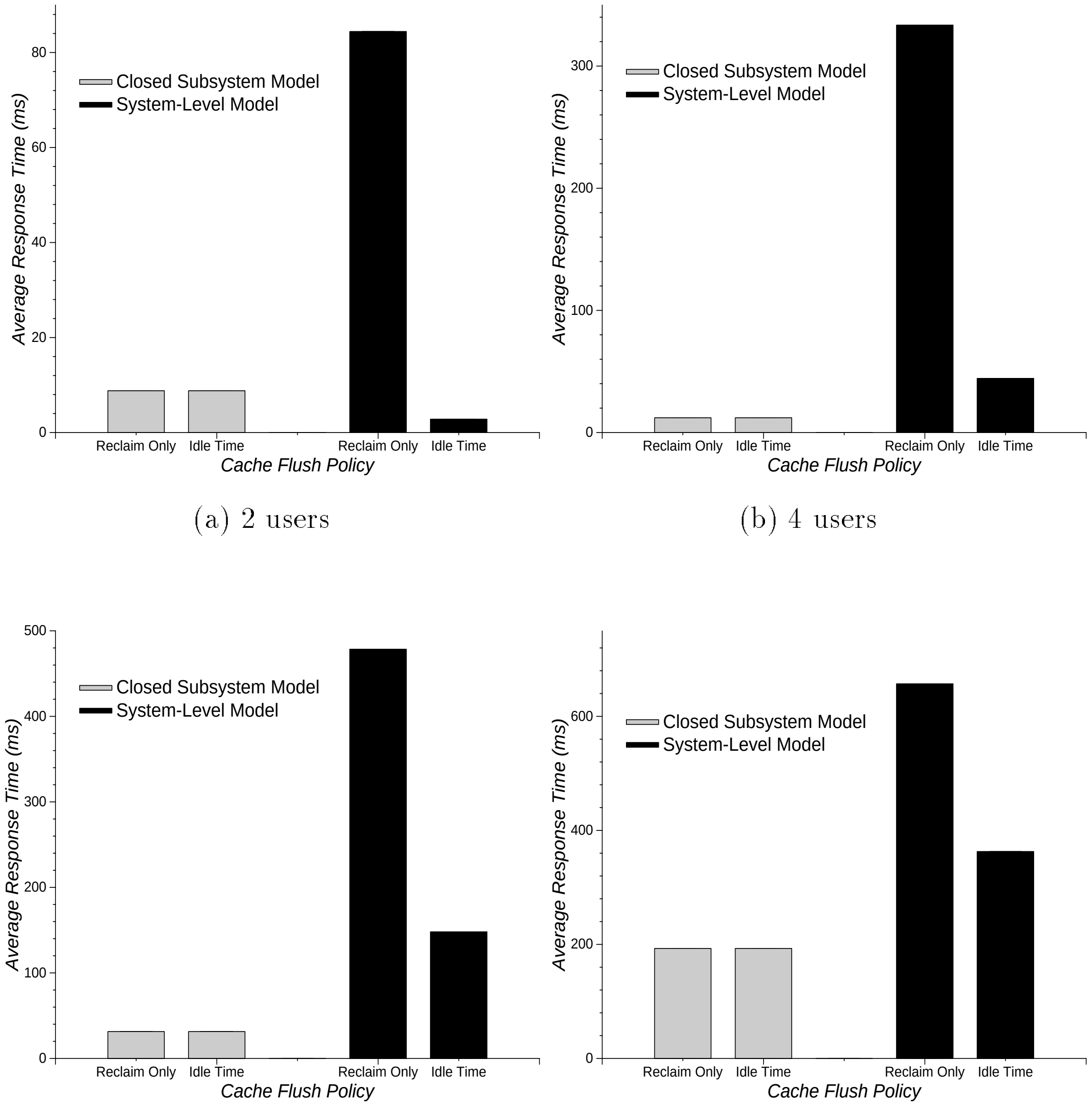
The closed subsystem model predicts no performance difference between the flush policies for any number of users. Because the closed subsystem model maintains a constant, non-zero number of requests outstanding at any point in time, there is no idle time to be exploited. A naive storage subsystem designer might observe these results and conclude that attempting to detect and use idle time is a waste of implementation effort, because it results in no performance improvement.

On the other hand, the system-level model correctly predicts that using idle time to flush dirty cache blocks improves performance significantly. The average response time reductions shown in figure 6.8 range from 45% to 97%. The improvement is larger with fewer users because there is more idle time to exploit. This example highlights a major problem with closed subsystem models, showing that they can lead to incorrect conclusions.

6.3.3 Cache-Aware Disk Scheduling

This subsection evaluates the use of disk cache awareness in aggressive disk scheduling algorithms located at the device driver, providing a non-obvious example where trivializing feedback effects leads to an erroneous conclusion. The **Shortest-Positioning-Time-First** (SPTF) algorithm uses full knowledge of processing overheads, logical-to-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 [Seltzer90, Jacobson91]. 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 [Worthington94]. In my experiments, the on-board disk cache services only read I/O requests because the HP C2247A disk drive used in the experimental system was configured to store data on the magnetic media before signaling completion for a write I/O request. The benefit of cache-awareness in the disk scheduler should depend upon the frequency with which read I/O requests that would hit in the on-board cache are otherwise delayed.

Giving preference to requests that hit in the on-board disk cache can improve SPTF performance in several ways. First, elementary queueing 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 outstanding sequential read requests in non-sequential 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



(a) 2 users

(b) 4 users

(c) 8 users

(d) 16 users

Figure 6.8: Disk Cache Flush Policy Comparison for SynRGen Workloads. The request populations for the closed subsystem models at 2 users, 4 users, 8 users and 16 users are 1, 1, 3 and 19, respectively.

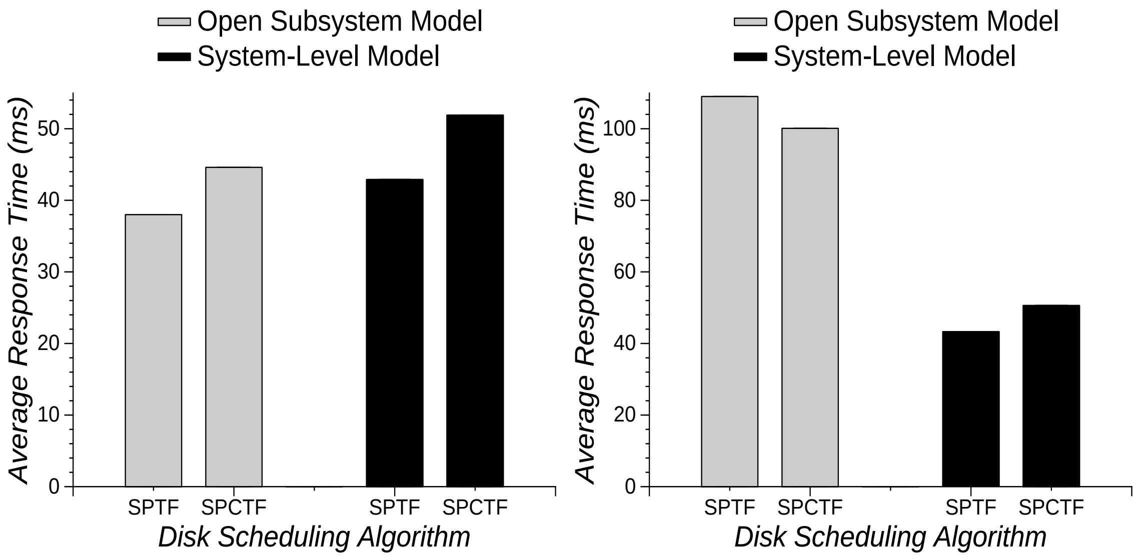
serviced is often not the next sequential request. During the bus transfer and completion processing of the first request, the media platter rotates past the beginning of the next sequential request (prefetching it, of course). The SPTF algorithm, which is ignorant of the prefetching, may not select the next sequential request. On the other hand, the SPCTF algorithm selects the next sequential request to exploit the prefetch behavior. Finally, SPCTF can improve performance when cache segments are re-used in a manner that reduces the number of cache hits. The HP C2247 uses one of its multiple cache segments exclusively for write requests. As a result, this problem arises only if there are multiple concurrent read request streams, which is not true of the workloads examined in this section.

Figures 6.9 and 6.10 compare SPTF and SPCTF with two different independent task workloads. Each figure contains two graphs, comparing average response times with scale factors of 1 and 2, respectively. Recall that the open subsystem model increases workload intensity by shrinking the measured request inter-arrival times. The system-level model scales a workload by increasing the speed of the host system components (CPU speed and memory bandwidth). Each graph shows two pairs of bars, comparing the average response times predicted with the open subsystem model and the system-level model, respectively.

In figure 6.9, the open subsystem model predicts that making the disk scheduler aware of the disk's on-board cache increases the average response time by 17% when using a trace scaling factor of one. This unexpected result is caused by interactions between the complex performance/workload feedback effects, which remain largely intact in this case, and the disk's prefetching behavior; this is explained further below. With a trace scaling factor of two, the open subsystem model predicts that the average response time decreases by 8%. The open subsystem model predicts similar behavior in figure 6.10 — a 0.6% increase with a scaling factor of one and a 6% decrease with a scale factor of two. Most of the performance improvement observed with the scale factor of two comes from servicing groups of outstanding sequential reads in ascending order, thereby exploiting the disk's prefetch behavior (as described above). A storage subsystem designer might observe these results and determine 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 in the figures, the closed subsystem model also predicts that SPCTF outperforms SPTF when disk scheduling affects performance. For the *compress* workload, SPCTF outperforms SPTF by 0.6% for request populations of four, eight, sixteen and thirty-two. For the *copytree* workload, SPCTF outperforms SPTF by 0.6% for a request population of thirty-two and by 0.2% for a population of sixteen requests. For populations of eight and four requests, no performance difference is observed. As with the open subsystem model, the performance improvement comes mainly from servicing sets of outstanding sequential reads in ascending order. While these average response time reductions are not large, a storage subsystem designer would be inclined to incorporate cache-awareness into the disk scheduler if the implementation effort is not unreasonable.

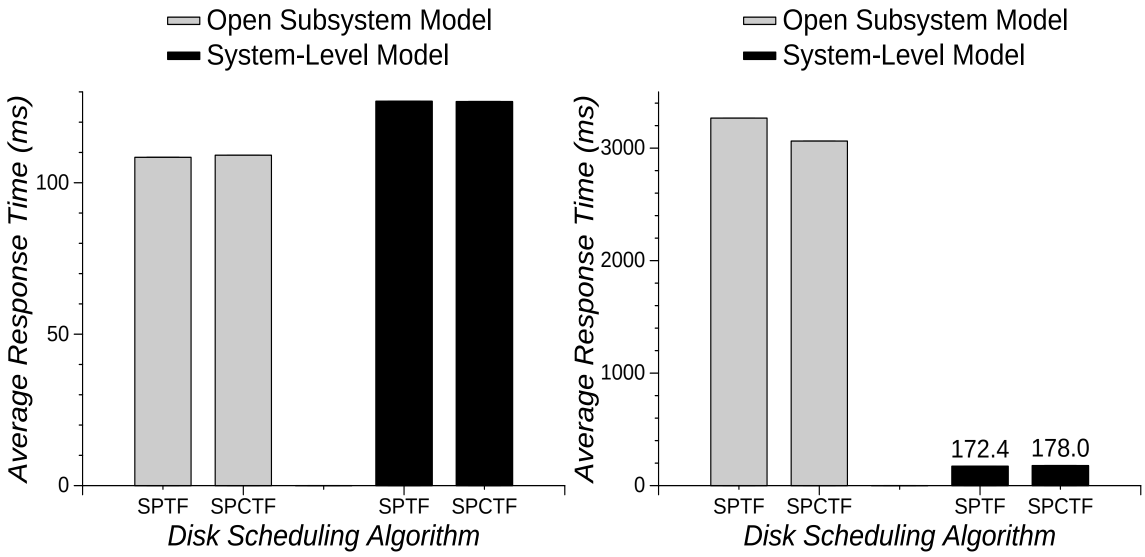
On the other hand, the system-level model predicts that cache-awareness in the disk scheduler consistently hurts storage subsystem performance. For the *compress* workload, the average response time increases by 21% with a scaling factor of one and by 17% with a scaling factor of two. For the *copytree* workload, there is no difference with a scaling factor of one, but the average response time increases by 3.2% with a scaling factor of two. The



(a) Scale Factor = 1.0

(b) Scale Factor = 2.0

Figure 6.9: Cache-Aware Disk Scheduling for the *compress* Workload.



(a) Scale Factor = 1.0

(b) Scale Factor = 2.0

Figure 6.10: Cache-Aware Disk Scheduling for the *copytree* Workload.

conclusion, given these performance predictions, is that SPTF is superior to SPCTF for these two workloads.[4]

Cache-awareness, as represented by the SPCTF scheduling algorithm, hurts performance because of complex interplay between performance/workload feedback effects and the disk's prefetching behavior. Most of the read I/O request arrivals in both of these workloads are caused by sequential file accesses and are therefore largely (but not entirely) sequential. The read I/O 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 block N. Therefore, the number of pending read I/O requests ranges between 0 and 2, greatly diminishing SPTF's difficulty with sets of outstanding sequential reads. Write I/O requests, arriving in bursts when the syncer daemon awakens to flush dirty file blocks, occasionally compete with these read I/O 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 0 pending read I/O requests and the disk scheduler (either variant) will initiate one of the write I/O requests.

At this point, SPTF and SPCTF part company. SPTF will continue to service the write I/O requests, independent of read I/O 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 I/O request(s) will wait for the entire group of writes to be serviced 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 disk will begin to re-position the read/write head in order to prefetch additional data. After servicing the read I/O 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 re-position to service the write. Frequently, the disk will not succeed in prefetching any data because re-positioning the read/write head requires more time than it takes to service a read from cache and begin the next request. This cycle (read hit, failed prefetch, write I/O request) may repeat several times before a new read I/O requests 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 re-positioning the read/write head for aborted prefetching activity decreases disk efficiency and increases the average response time.

Given that initiating internal disk prefetch activity after servicing a read request from the cache causes this performance problem, one might consider not doing so. However, this policy does improve performance and should remain in place. For example, for the *compress* workload with a scaling factor of one, the system-level model predicts that SPTF performance drops by 3% 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%, which is not enough to compensate for the elimination of cache hit prefetching. Among the cross-product of these options, the best storage subsystem performance (for these workloads) is offered by SPTF scheduling and aggressive prefetching after cache hits.

The interactions that cause cache-awareness are obscure and easy to overlook without some indication that they represent a problem. [Worthington94] compared SPCTF and

[4] For the example in the next chapter, increased request response times do not translate into increased elapsed times for the tasks. In this example, they do.

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 non-obvious ways.

6.4 Summary

The conventional methodology fails to accurately model feedback effects between storage subsystem performance and the workload. This chapter demonstrates that this shortcoming can lead to both quantitative and qualitative errors. Open subsystem models tend to over-estimate performance changes and often allow unrealistic concurrency in the workload. Closed subsystem models tend to under-estimate performance changes and completely ignore burstiness in request arrival patterns. The unrealistic concurrency and lack of burstiness can lead directly to erroneous conclusions in open and closed subsystem models, respectively. Less direct interactions between complex performance/workload feedback and storage subsystem behavior can also lead conventional methodology to erroneous conclusions that are very difficult to recognize without a more accurate representation of feedback.

CHAPTER 7

Criticality-Based Disk Scheduling

Even when storage subsystem performance is correctly predicted, the conventional subsystem-oriented methodology can promote sub-optimal 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). This chapter demonstrates this problem by evaluating a storage subsystem modification that improves overall system performance while reducing performance as measured by subsystem metrics. The chapter describes the sources of request criticality in the experimental system and shows measurements of criticality mixtures in different workloads. The system-level simulation model is used to evaluate criticality-based disk scheduling, which gives preference to system needs rather than mechanical positioning delays. The system-level model reports overall system performance metrics as well as storage subsystem metrics. The system metrics show an increase in performance while the storage subsystem metrics indicate a decrease in performance.

7.1 Request Criticality

I/O requests can be partitioned into three request criticality classes based on how they interact with executing processes. A request is time-critical if the process that generates it, either explicitly or implicitly, must stop executing until the request is complete. Time-limited requests are those that become time-critical if not completed within some amount of time (called the time limit). Finally, time-noncritical denotes a request that no application process waits for. It must be completed within some reasonable amount of time to maintain the accuracy of the non-volatile storage, to free system resources (e.g., memory) held on their behalf, and/or to allow background activity to progress.

The remainder of this section describes the sources of requests belonging to the three classes in our experimental system and presents measurement data regarding the criticality mixes present in different workloads.

7.1.1 Sources of Request Criticality

In the experimental system, storage I/O requests can be generated directly by three system software components: the file system, the virtual memory management system and the direct I/O system call interface. No requests were generated by the latter two components in any of the workloads for which system-level traces were captured. That is, the file system was responsible for all requests in these workloads. As mentioned in section 5.3, the experiments were run using the *ufs* file system.

All read I/O requests generated by the *ufs* file system can be characterized as either demand fetches or prefetches. Demand fetches (i.e., file block cache misses) are time-critical and useful prefetches are time-limited. I/O requests that prefetch unnecessary data are completely useless and are therefore classified as time-noncritical. The *ufs* file system prefetches file blocks sequentially until non-sequential access is detected. Measurements of UNIX file systems indicate that most user-level file reads are sequential [Ousterhout85, Baker91]. This is true of the system-level workloads used in this dissertation.

The *ufs* file system distinguishes between three types of file block writes: synchronous, asynchronous and delayed [Ritchie86]. A **synchronous** file write immediately generates the corresponding I/O request and causes the process to block and wait until the request completes. So, synchronous file writes result in time-critical I/O requests. An **asynchronous** file write also immediately generates the corresponding write I/O request, but does not wait for the request to complete. An I/O request generated by an asynchronous file write is either time-limited or time-noncritical, depending upon access locality and the locking policy. If a source memory block is locked for the duration of an I/O request and a process (the originator or another process) subsequently attempts to access it, then the I/O request becomes time-critical (meaning that it was time-limited). Otherwise, it is time-noncritical. A **delayed** file write dirties one or more blocks in the file block cache but does not immediately generate a write I/O request. Write I/O requests for dirty cache blocks are generated in the background by the file system. The lock conflict scenario described above can occur, but the majority of these requests are time-noncritical.[1]

In the *ufs* file system, delayed writes are used for updating all normal file data unless otherwise specified by the user. During write I/O requests, dirty file blocks from normal files are not locked. Synchronous and asynchronous writes are used for many file system metadata updates in order to enforce update sequencing constraints (to maintain file system integrity).[2] In addition, metadata blocks are locked for the duration of the corresponding write I/O requests. Because these metadata blocks exhibit high degrees of update locality [Ganger94], most of the asynchronous metadata writes result in time-limited I/O requests. The experimental system contains a reasonable quantity of physical main memory and therefore does not experience write-thrashing problems unless the workload consists of a very large file working set.

[1] If the file block cache is too small to handle the working set of written file data, then most write I/O requests will become time-critical or time-limited because dirty blocks must be cleaned before they can be re-used. Measurements of real systems suggest that this should be a rare occurrence [Ruemmler93a]. The exception would be when the main memory capacity is too small to maintain a balanced system (e.g., [Bennett94]).

[2] This use of synchronous and asynchronous writes for metadata update sequencing represents a significant file system performance problem. Aggressive implementation techniques can eliminate them [Hagmann87, Seltzer93, Ganger94].

HP-UX	Read Requests		Write Requests	
Trace	Critical	Limited/Noncritical	Critical	Limited/Noncritical
cello	36.4 %	7.7 %	37.2 %	18.7 %
snake	37.9 %	5.9 %	13.6 %	42.6 %
hplajw	38.4 %	4.0 %	28.8 %	28.8 %

Table 7.1: Request Criticality Breakdown for HP-UX Traces. This data was taken from [Ruemmler93]. The traces are described briefly in section A.2.2.1 and thoroughly in [Ruemmler93]. Information distinguishing between time-limited and time-noncritical requests is not available in the traces.

System-Level	Read Requests			Write Requests		
Workload	Critical	Limited	Noncritical	Critical	Limited	Noncritical
compress	0.9 %	69.6 %	–	0.3 %	–	29.2 %
uncompress	1.1 %	24.9 %	–	0.3 %	–	73.7 %
copytree	7.8 %	36.1 %	–	6.8 %	2.0 %	47.3 %
removetree	1.4 %	0.2 %	–	93.4 %	–	4.9 %
synrgen1	1.8 %	2.5 %	–	25.8 %	2.1 %	67.8 %
synrgen2	2.8 %	3.1 %	–	25.5 %	2.0 %	66.6 %
synrgen4	3.6 %	6.8 %	–	23.0 %	1.9 %	64.7 %
synrgen8	4.6 %	10.5 %	–	24.8 %	2.1 %	57.9 %
synrgen16	8.8 %	11.6 %	–	25.7 %	2.8 %	51.0 %

Table 7.2: Request Criticality Breakdown for System-Level Traces. These data are from system-level traces of the workloads described in section 5.2 executing on the experimental system.

7.1.2 Measurements of Request Criticality

Most real workloads contain requests from all three request criticality classes. Table 7.1 shows data regarding the criticality mixes observed during extensive measurements of three different HP-UX systems [Ruemmler93]. Table 7.2 shows more precise data extracted from the system-level traces described in section 5.2. The data in these tables demonstrate both that criticality mixes are common and that the exact mixture varies widely between workloads.

An important aspect of how time-limited requests impact overall system performance is the duration of the time limits. The time limit determines how long an I/O subsystem can take to service a request without causing any process to block. Long time limits allow an I/O subsystem significant latitude in when and how to service requests. Figures 7.1–7.6 show time limit densities for the workloads described in section 5.2.

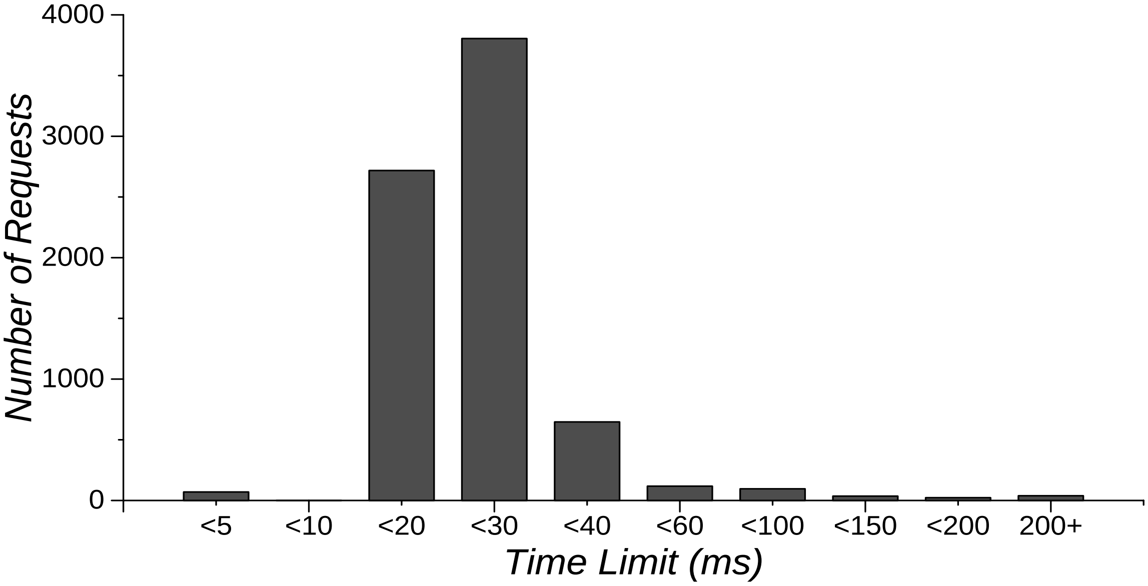


Figure 7.1: Time Limit Density for the *compress* Workload. The average time limit is 25.8 ms, and the maximum is 611 ms.

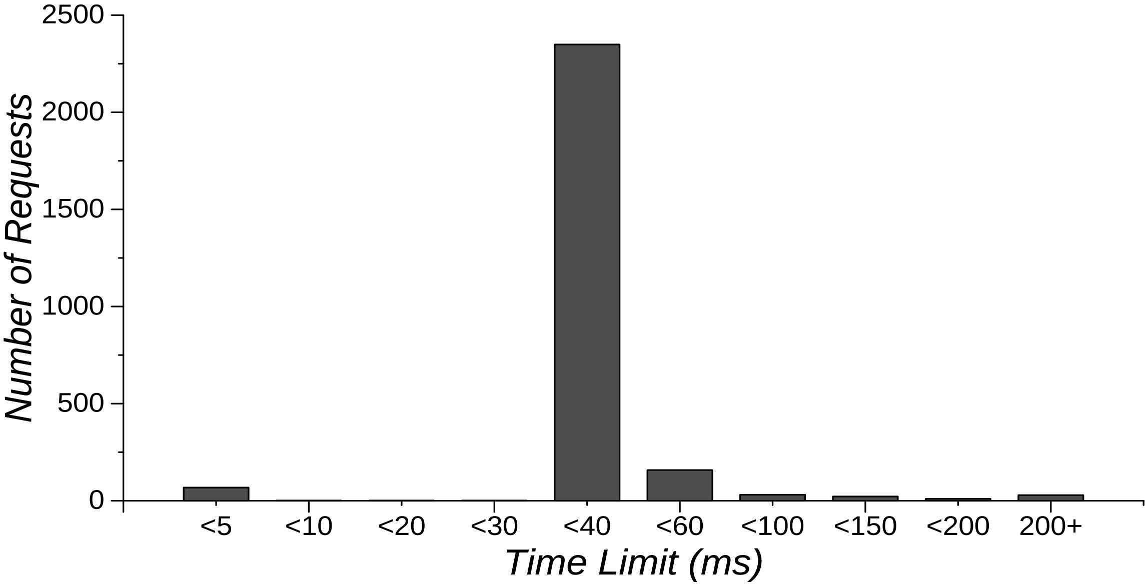


Figure 7.2: Time Limit Density for the *uncompress* Workload. The average time limit is 111 ms, and the maximum is 144 seconds.

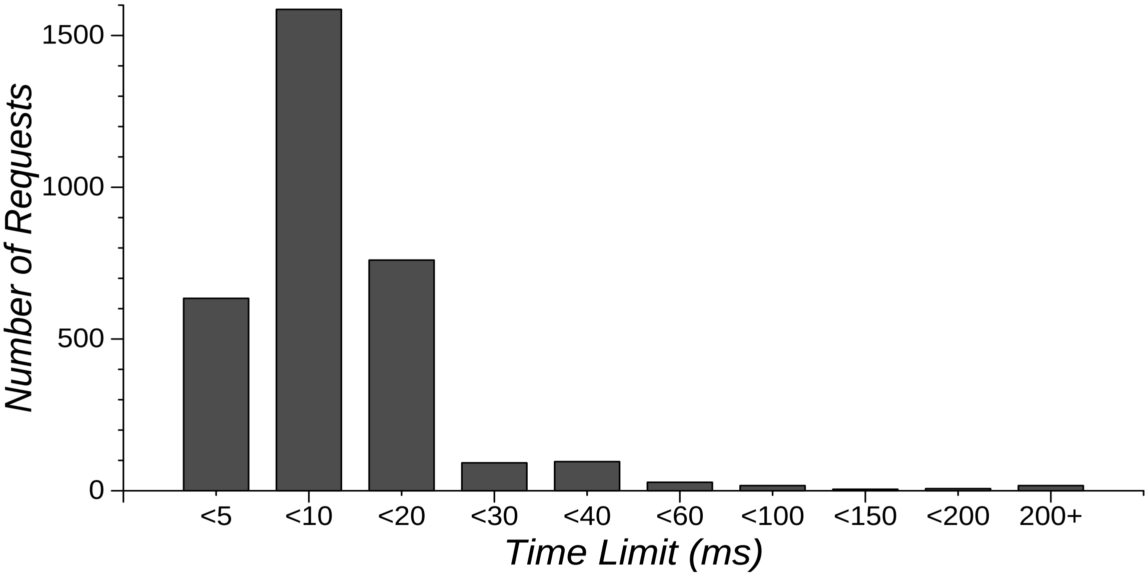


Figure 7.3: Time Limit Density for the *copytree* Workload. The average time limit is 60.1 ms, and the maximum is 155 seconds.

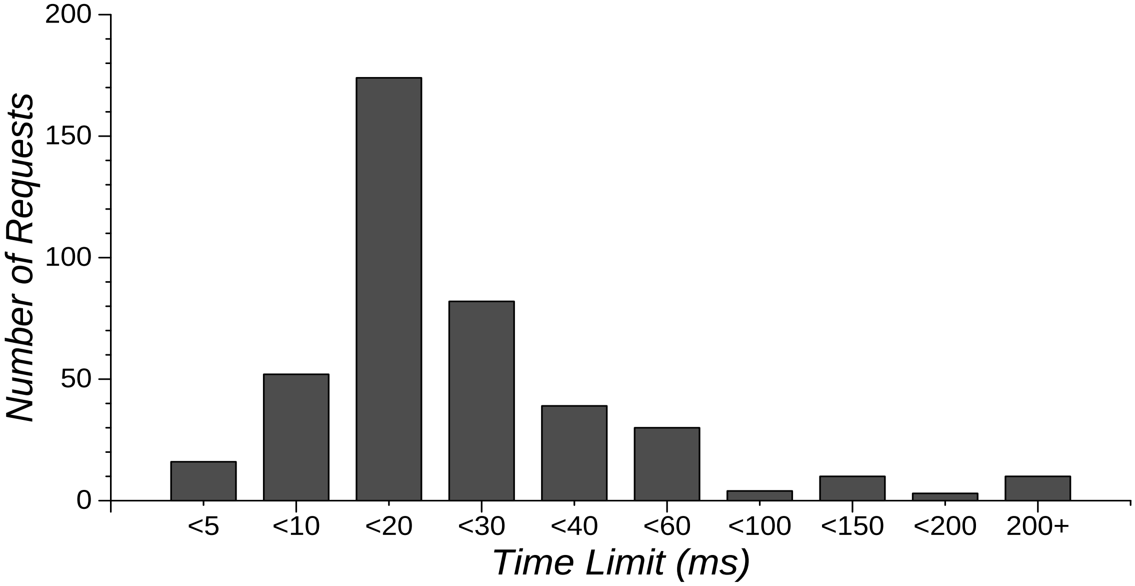


Figure 7.4: Time Limit Density for the *synrgen16* Workload. The average time limit is 51.7 ms, and the maximum is 3.18 seconds.

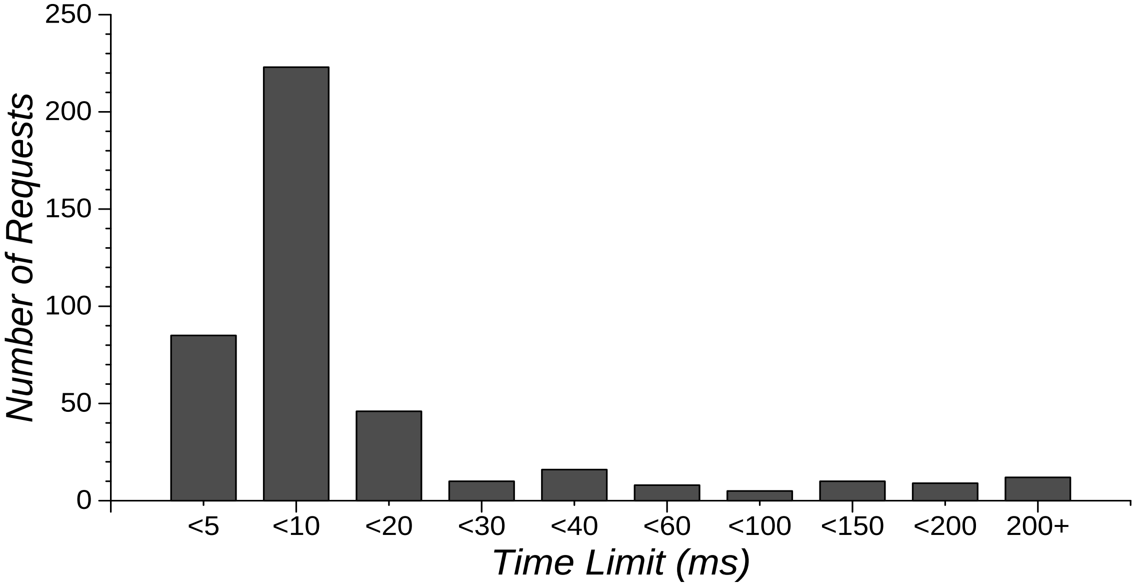


Figure 7.5: Time Limit Density for the *synrgen8* Workload. The average time limit is 24.8 ms, and the maximum is 493 ms.

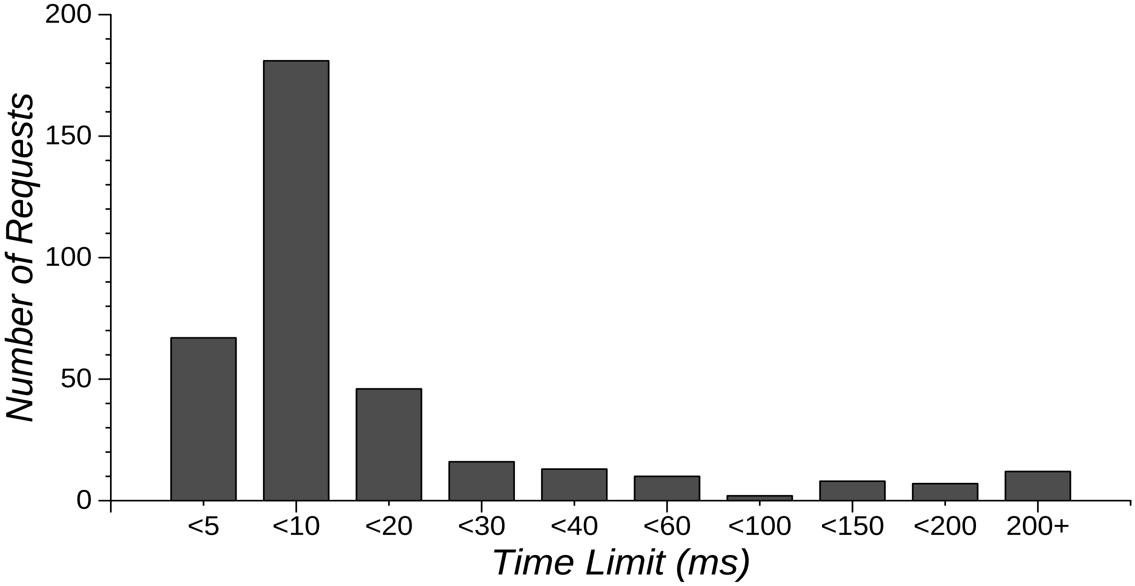


Figure 7.6: Time Limit Density for the *synrgen4* Workload. The average time limit is 52.6 ms, and the maximum is 5.96 seconds.

7.2 Disk Request Scheduling Algorithms

Many different disk scheduling algorithms can be used to reduce mechanical positioning delays. This chapter focuses on two of them. 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 last one, then the request with the lowest starting block number is serviced. I use the former algorithm because it performs no storage subsystem performance optimizations. The latter is used because it has reasonable implementation costs, is used in many existing systems, and has been shown to outperform other seek-reducing algorithms for many real workloads [Worthington94].

In addition to these conventional algorithms, this chapter examines disk scheduling algorithms that prioritize requests based on request criticality. From a short-term viewpoint, time-critical and time-limited are clearly more important to system performance than time-noncritical requests. This chapter evaluates the performance impact of modifying each of the base algorithms to maintain two distinct lists of pending requests and always servicing requests from the higher-priority list first.[3] 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 the measurement data above indicate that they often have very short time limits. The scheduling criteria used for each list remain the same as with the base algorithm.

To implement this algorithm, the disk request scheduler must have per-request criticality information. Fortunately, the file system (as well as the other request generating components) generally knows the criticality class to which a request belongs when it is generated. Requests caused by demand fetches and synchronous metadata updates are time-critical. Background writes of dirty file blocks are time-noncritical. Requests resulting from prefetches and asynchronous metadata updates are usually time-limited. In the implementation of this algorithm, I assume that all such requests are time-limited. The file system was modified to pass this information to the device driver with each request as part of the I/O control block.

More aggressive algorithms might utilize knowledge of time limit durations, since completing a request just before its time-limit expires is no worse than completing it in zero time (from the associated process's point of view). Unfortunately, identifying time limits in advance is nontrivial. For example, the time limit for a prefetch depends upon how much computation the associated process can do before the data are required, as well as any interrupt or context switching activity. Lock-related time limits are even more difficult to predict because the process that waits for the request is often not be the process that generated it. Therefore, time limits can only be estimated in most systems. The estimates should be as high as possible to allow latitude in scheduling decisions. At the same time, they should be as low as necessary to avoid the penalties associated with failing to complete requests within their actual time limits.

[3] Many variations and more aggressive algorithms can be devised, but, as stated earlier, my goal is not to fully explore the design space of disk scheduling algorithms that use criticality information. Rather, my intent is to demonstrate a problem with storage subsystem models, as well as indicate the potential of this performance enhancement.

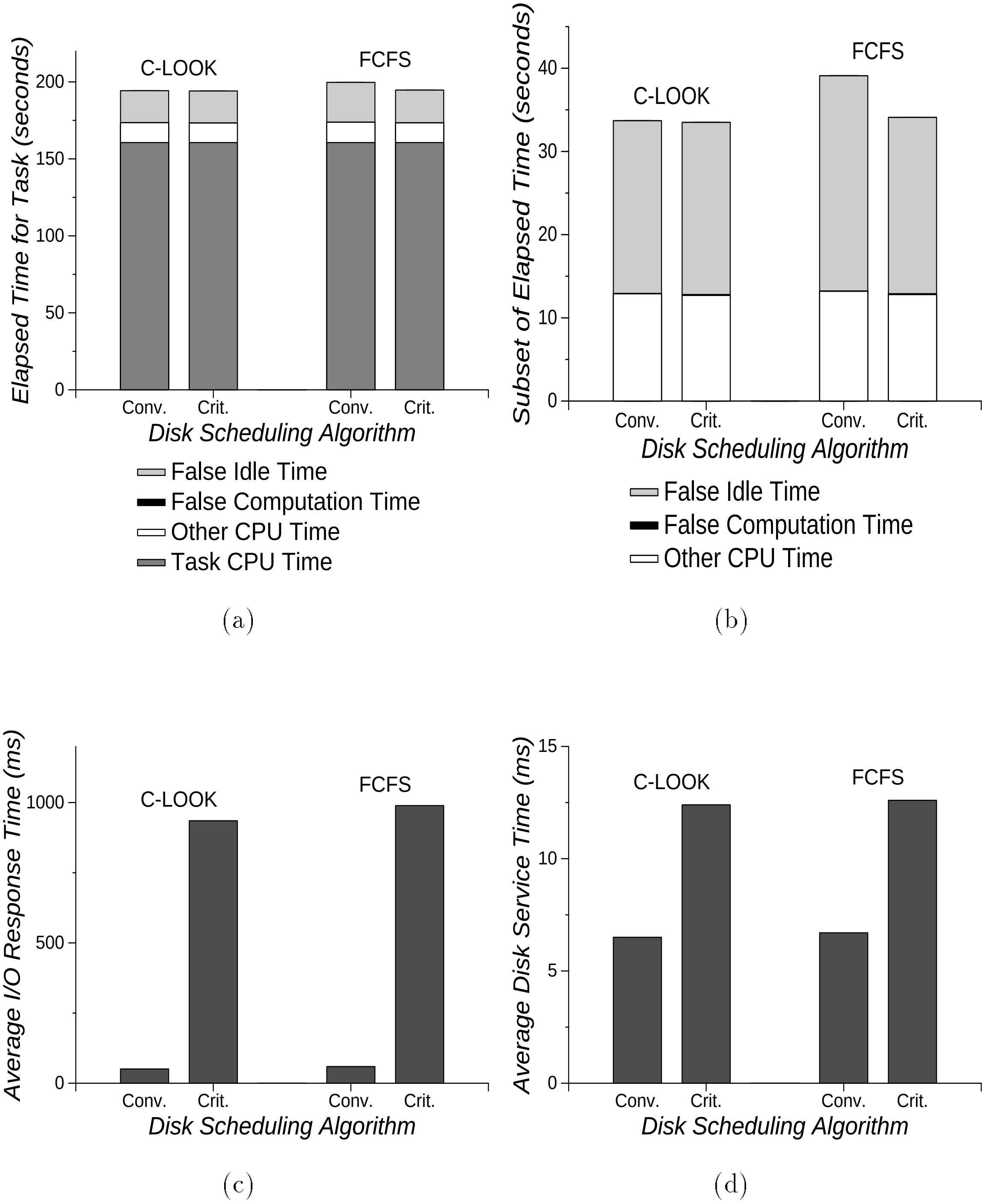
7.3 Performance Comparison

Figures 7.7–7.12 compare four disk scheduling algorithms in terms of both system performance and storage subsystem performance. Each figure contains four graphs labeled (a)–(d). Each graph contains four bars, representing conventional and criticality-based versions of C-LOOK and FCFS. In each figure, graph (a) shows overall system performance as measured by the elapsed time for a user task. The elapsed times are broken into four regions, indicating the causes of different portions of the elapsed time. The particular regions shown vary among the workloads (see below). Graph (b) shows a subset of the same data, highlighting the portions affected by storage subsystem performance by removing a region of the elapsed time that is independent of subsystem performance (see below). Graphs (c) and (d) show the average response time and average service time across all requests, respectively. Tables 7.3–7.8 provide additional information to assist in understanding the data in the figures.

Graphs (a) and (c) provide commonly utilized performance metrics for the overall system and the storage subsystem, respectively. For all of the workloads shown, overall system performance increases when the disk scheduler uses request criticality information. At the same time, storage subsystem performance decreases substantially (over 2 orders of magnitude in some cases). Most of the overall system performance improvement comes from reducing false idle time by expediting the completion of requests that cause processes to block. 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.

It is interesting to note that, when enhanced with criticality information, FCFS and C-LOOK result in roughly equivalent overall system performance for the workloads tested. This differs from the behavior noted for the conventional algorithms, where C-LOOK provides improved system performance. This change in behavior is not overly surprising given the nature of how the criticality-based enhancement operates and how different requests affect system performance. Pending time-critical and time-limited requests are kept in one list and time-noncritical requests in another. The former list is almost always very short because time-critical and time-limited requests tend to block processes, preventing them from generating additional requests. It is well-established that disk scheduling algorithms all perform equally for consistently small lists of pending requests [Teorey72, Worthington94]. On the other hand, the second list, which contains the time-noncritical requests, tends to grow large and presents solid opportunities for improvement. Indeed, the results show that C-LOOK results in lower average response times for these requests. However, unless the system becomes bottlenecked by the storage subsystem (which does not occur with these workloads), the response times of time-noncritical requests are not important to overall system performance. Of course, reducing time-noncritical request service times increases the ability of the storage subsystem to avoid becoming a system performance bottleneck.

The remainder of this section explains these performance effects in more detail, focusing first on the individual task workloads and then on the SynRGen workloads.



(a)

(b)

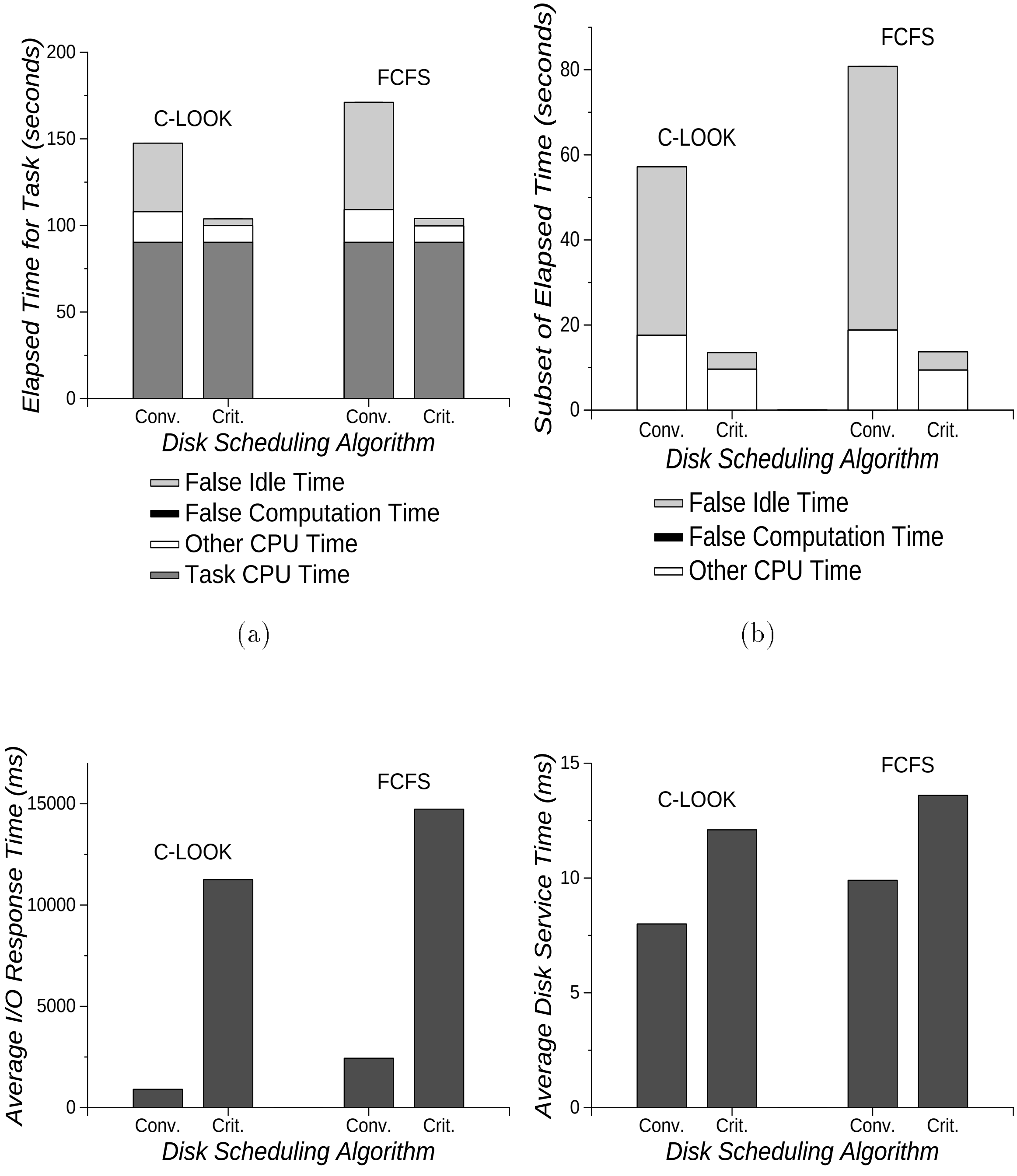
(c)

(d)

Figure 7.7: Criticality-Based Scheduling of the *compress* Workload.

Performance Metric	C-LOOK		FCFS	
	Conv.	Criticality	Conv.	Criticality
Elapsed Time for Task	194.3 sec	194.1 sec	199.7 sec	194.7 sec
Task Computation Time	160.6 sec	160.6 sec	160.6 sec	160.6 sec
Other CPU Time	12.9 sec	12.7 sec	13.2 sec	12.8 sec
False Computation Time	0.03 sec	0.1 sec	0.03 sec	0.1 sec
Task I/O Wait Time	23.8 sec	22.5 sec	29.3 sec	23.1 sec
False Idle Time	20.8 sec	20.7 sec	26.0 sec	21.3 sec
Time-Critical Requests	132	132	132	132
Avg. Response Time	25.3 ms	22.6 ms	27.5 ms	22.8 ms
Max. Response Time	343 ms	86.0 ms	551 ms	78.5 ms
Time-Limited Requests	7567	7567	7567	7567
Avg. Response Time	10.4 ms	15.3 ms	11.7 ms	15.4 ms
Max. Response Time	643 ms	92.5 ms	926 ms	80.5 ms
Avg. Time Limit	28.5 ms	22.2 ms	33.6 ms	22.3 ms
% Satisfied in Time	96.0 %	76.4 %	97.4 %	76.1 %
Time-Noncritical Reqs.	3171	3171	3171	3171
Avg. Response Time	148 ms	3169 ms	173 ms	3353 ms
Max. Response Time	708 ms	27.5 sec	940 ms	16.3 sec
Avg. Service Time	6.5 ms	12.4 ms	6.7 ms	12.6 ms
% Disk Buffer Hits	63.5 %	32.9 %	64.8 %	32.7 %
Avg. Seek Distance	124 cyls	558 cyls	126 cyls	562 cyls
Avg. Seek Time	3.5 ms	8.5 ms	4.5 ms	8.7 ms

Table 7.3: Criticality-Based Scheduling of the *compress* Workload.



(a)

(b)

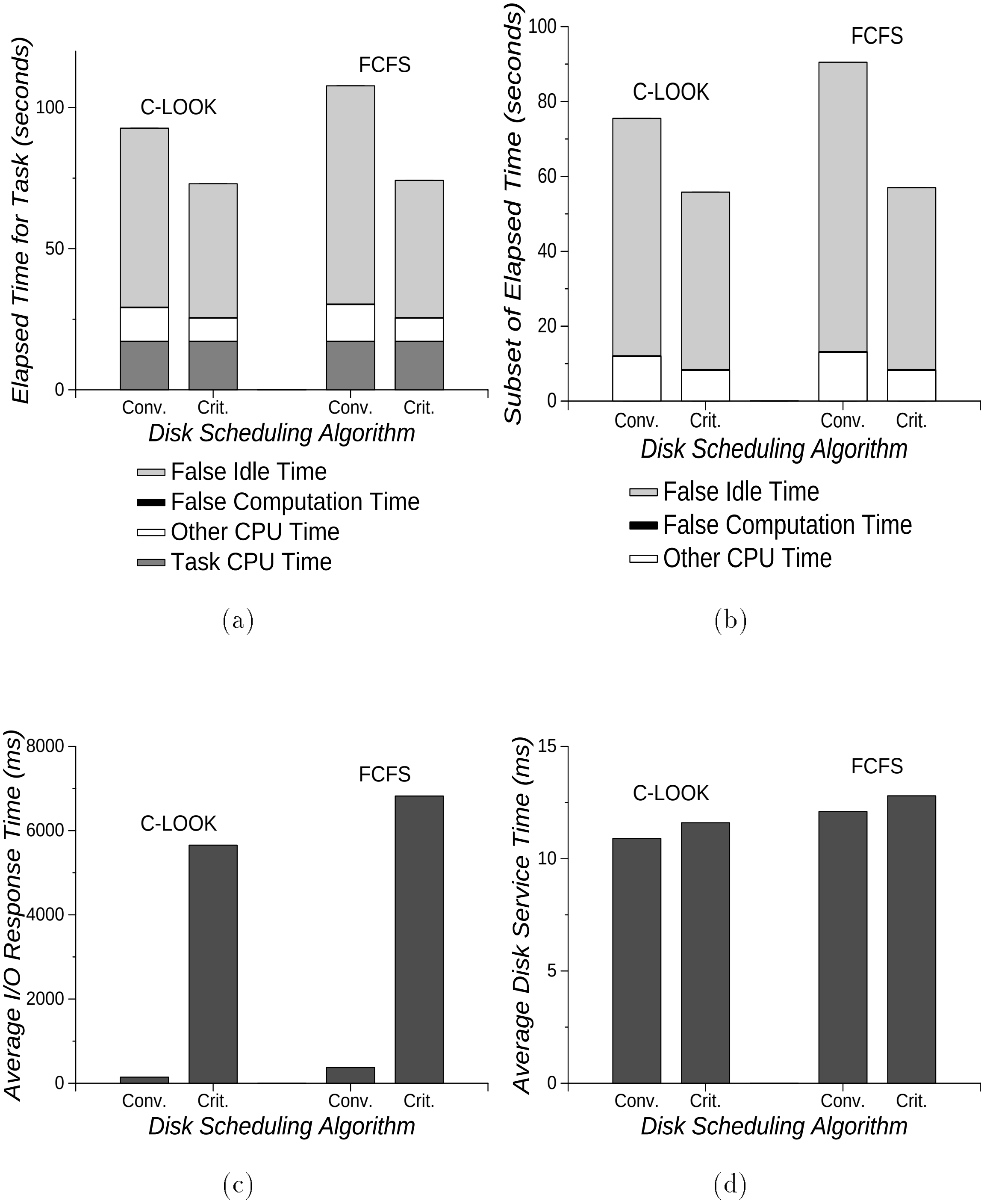
(c)

(d)

Figure 7.8: Criticality-Based Scheduling of the *uncompress* Workload.

Performance Metric	C-LOOK		FCFS	
	Conv.	Criticality	Conv.	Criticality
Elapsed Time for Task	147.5 sec	103.8 sec	171.1 sec	104.0 sec
Task Computation Time	90.3 sec	90.3 sec	90.3 sec	90.3 sec
Other CPU Time	17.6 sec	9.6 sec	18.8 sec	9.4 sec
False Computation Time	0.02 sec	0.03 sec	0.02 sec	0.03 sec
Task I/O Wait Time	51.0 sec	4.3 sec	75.9 sec	4.8 sec
False Idle Time	39.6 sec	3.9 sec	62.0 sec	4.3 sec
Time-Critical Requests	153	153	153	153
Avg. Response Time	62.7 ms	29.1 ms	190.6 ms	30.2 ms
Max. Response Time	6165 ms	95.1 ms	10.2 sec	98.0 ms
Time-Limited Requests	2693	2693	2693	2693
Avg. Response Time	42.5 ms	19.3 ms	59.0 ms	19.9 ms
Max. Response Time	11.0 sec	73.6 ms	10.2 sec	91.6 ms
Avg. Time Limit	31.0 ms	33.8 ms	33.6 ms	34.2 ms
% Satisfied in Time	95.9 %	86.5 %	97.3 %	84.8 %
Time-Noncritical Reqs.	7989	7989	7989	7989
Avg. Response Time	1207 ms	15.3 sec	3258 ms	20.0 sec
Max. Response Time	10.5 sec	69.7 sec	12.1 sec	62.2 sec
Avg. Service Time	8.0 ms	12.1 ms	9.9 ms	13.6 ms
% Disk Buffer Hits	22.4 %	8.2 %	22.5 %	8.0 %
Avg. Seek Distance	31 cyls	217 cyls	50 cyls	253 cyls
Avg. Seek Time	1.5 ms	4.2 ms	2.8 ms	5.6 ms

Table 7.4: Criticality-Based Scheduling of the *uncompress* Workload.



(a)

(b)

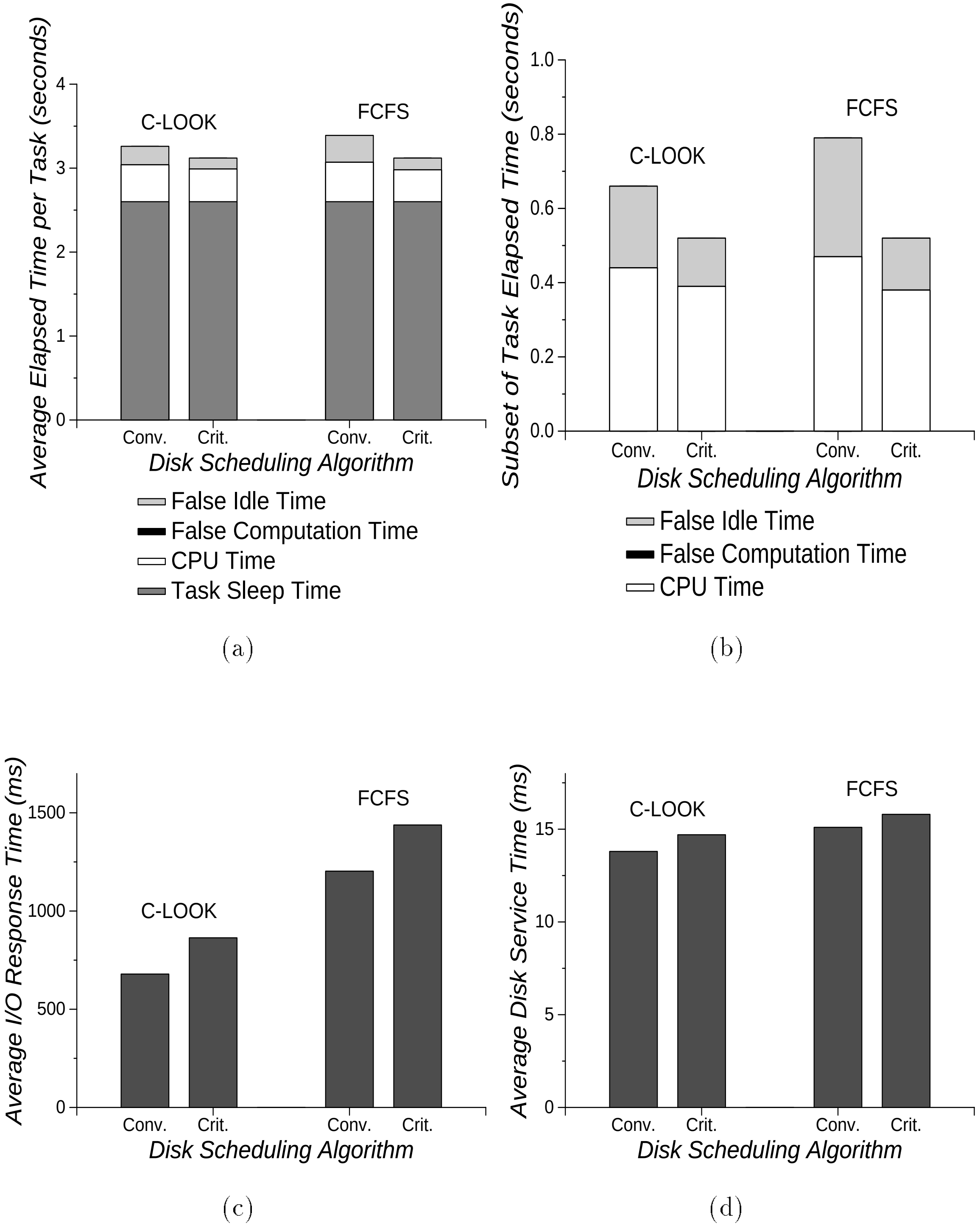
(c)

(d)

Figure 7.9: Criticality-Based Scheduling of the *copytree* Workload.

Performance Metric	C-LOOK		FCFS	
	Conv.	Criticality	Conv.	Criticality
Elapsed Time for Task	92.7 sec	73.0 sec	107.7 sec	74.2 sec
Task Computation Time	17.2 sec	17.2 sec	17.2 sec	17.2 sec
Other CPU Time	11.9 sec	8.2 sec	13.0 sec	8.2 sec
False Computation Time	0.2 sec	0.2 sec	0.2 sec	0.2 sec
Task I/O Wait Time	72.1 sec	52.8 sec	87.4 sec	53.9 sec
False Idle Time	63.4 sec	47.5 sec	77.3 sec	48.6 sec
Time-Critical Requests	1502	1502	1502	1502
Avg. Response Time	43.8 ms	30.1 ms	46.9 ms	29.9 ms
Max. Response Time	2040 ms	126 ms	5039 ms	99.3 ms
Time-Limited Requests	3270	3270	3270	3270
Avg. Response Time	20.6 ms	15.2 ms	27.6 ms	15.6 ms
Max. Response Time	1629 ms	130 ms	5650 ms	93.7 ms
Avg. Time Limit	11.8 ms	10.2 ms	13.9 ms	10.3 ms
% Satisfied in Time	2.6 %	2.6 %	2.6 %	2.4 %
Time-Noncritical Reqs.	4279	4279	4279	4279
Avg. Response Time	274.9 ms	11.9 sec	750.7 ms	14.4 sec
Max. Response Time	2321 ms	60.3 sec	6410 ms	33.2 sec
Avg. Service Time	10.9 ms	11.6 ms	12.1 ms	12.8 ms
% Disk Buffer Hits	1.6 %	1.2 %	1.9 %	1.1 %
Avg. Seek Distance	104 cyls	124 cyls	142 cyls	173 cyls
Avg. Seek Time	2.9 ms	3.2 ms	4.0 ms	4.6 ms

Table 7.5: Criticality-Based Scheduling of the *copytree* Workload.



(a)

(b)

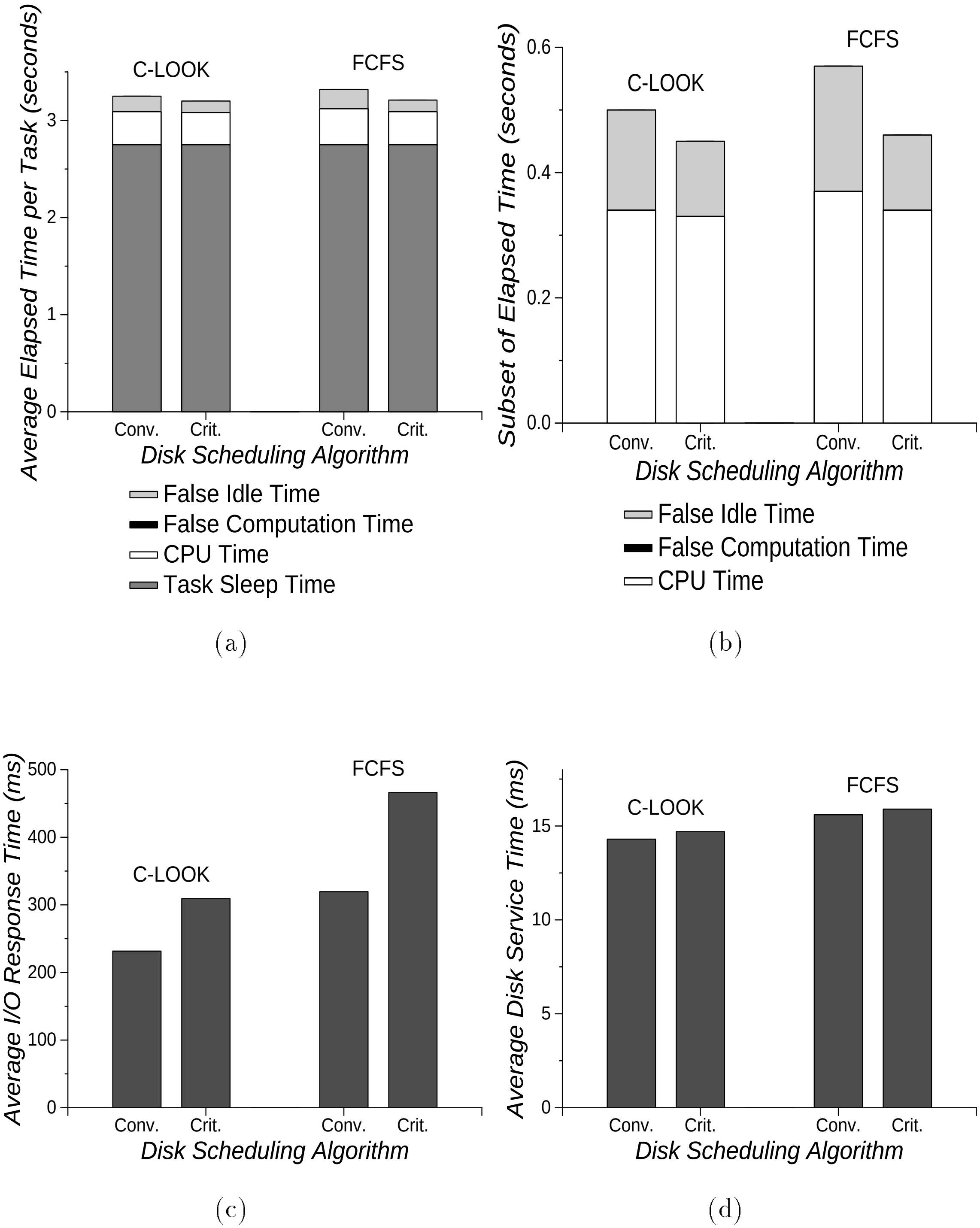
(c)

(d)

Figure 7.10: Criticality-Based Scheduling of the *synrgen16* Workload.

Performance Metric	C-LOOK		FCFS	
	Conv.	Criticality	Conv.	Criticality
Avg. Total Task Time	3.26 sec	3.12 sec	3.39 sec	3.12 sec
Avg. Task Sleep Time	2.60 sec	2.60 sec	2.60 sec	2.60 sec
Avg. Task False Comp.	0.01 sec	0.01 sec	0.01 sec	0.01 sec
Avg. Task I/O Wait	0.33 sec	0.19 sec	0.45 sec	0.19 sec
Avg. Task False Idle	0.22 sec	0.13 sec	0.32 sec	0.14 sec
Time-Critical Requests	1813	1813	1824	1813
Avg. Response Time	68.2 ms	26.9 ms	100.7 ms	27.0 ms
Max. Response Time	18.0 sec	335 ms	19.1 sec	172 ms
Time-Limited Requests	606	608	609	609
Avg. Response Time	43.5 ms	24.9 ms	83.4 ms	28.7 ms
Max. Response Time	9797 ms	269 ms	14.6 sec	169 ms
Avg. Time Limit	25.4 ms	11.2 ms	29.5 ms	11.6 ms
% Satisfied in Time	28.5 %	26.6 %	28.5 %	26.1 %
Time-Noncritical Reqs.	3179	2826	3108	2805
Avg. Response Time	1149 ms	1581 ms	2070 ms	2657 ms
Max. Response Time	18.7 sec	31.4 sec	20.8 sec	28.0 sec
Avg. Service Time	13.8 ms	14.7 ms	15.1 ms	15.8 ms
% Disk Buffer Hits	0.6 %	0.4 %	0.6 %	0.4 %
Avg. Seek Distance	122 cyls	191 cyls	204 cyls	251 cyls
Avg. Seek Time	4.5 ms	5.6 ms	6.0 ms	6.6 ms

Table 7.6: Criticality-Based Scheduling of the *synrgen16* Workload.



(a)

(b)

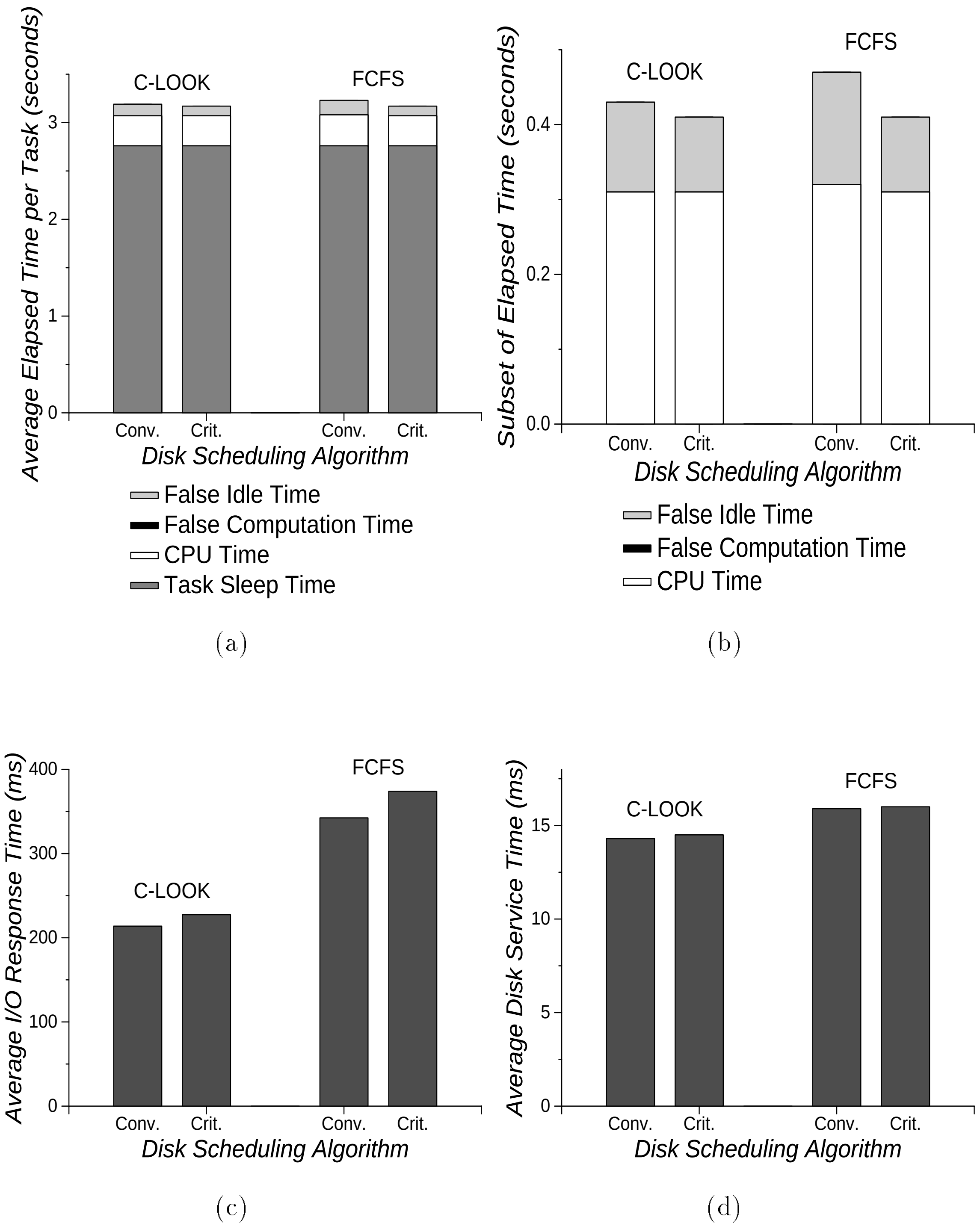
(c)

(d)

Figure 7.11: Criticality-Based Scheduling of the *synrgen8* Workload.

Performance Metric	C-LOOK		FCFS	
	Conv.	Criticality	Conv.	Criticality
Avg. Total Task Time	3.25 sec	3.20 sec	3.32 sec	3.21 sec
Avg. Task Sleep Time	2.75 sec	2.75 sec	2.75 sec	2.75 sec
Avg. Task False Comp.	0.01 sec	0.01 sec	0.01 sec	0.01 sec
Avg. Task I/O Wait	0.20 sec	0.14 sec	0.26 sec	0.15 sec
Avg. Task False Idle	0.16 sec	0.12 sec	0.20 sec	0.12 sec
Time-Critical Requests	1619	1648	1618	1648
Avg. Response Time	35.1 ms	21.7 ms	44.5 ms	22.0 ms
Max. Response Time	7794 ms	267 ms	7617 ms	223 ms
Time-Limited Requests	581	588	578	588
Avg. Response Time	25.2 ms	20.6 ms	35.5 ms	22.4 ms
Max. Response Time	3001 ms	214 ms	4659 ms	122 ms
Avg. Time Limit	12.2 ms	9.9 ms	11.8 ms	11.8 ms
% Satisfied in Time	32.3 %	29.2 %	32.0 %	28.6 %
Time-Noncritical Reqs.	3365	3353	3389	3366
Avg. Response Time	362 ms	501 ms	499 ms	761 ms
Max. Response Time	8045 ms	9338 ms	7649 ms	9513 ms
Avg. Service Time	14.3 ms	14.7 ms	15.6 ms	15.9 ms
% Disk Buffer Hits	1.0 %	0.7 %	1.1 %	0.7 %
Avg. Seek Distance	148 cyls	175 cyls	226 cyls	248 cyls
Avg. Seek Time	5.2 ms	5.6 ms	6.5 ms	6.8 ms

Table 7.7: Criticality-Based Scheduling of the *synrgen8* Workload.



(a)

(b)

(c)

(d)

Figure 7.12: Criticality-Based Scheduling of the *synrgen4* Workload.

Performance Metric	C-LOOK		FCFS	
	Conv.	Criticality	Conv.	Criticality
Avg. Total Task Time	3.19 sec	3.17 sec	3.23 sec	3.17 sec
Avg. Task Sleep Time	2.76 sec	2.76 sec	2.76 sec	2.76 sec
Avg. Task False Comp.	0.01 sec	0.01 sec	0.01 sec	0.01 sec
Avg. Task I/O Wait	0.14 sec	0.11 sec	0.17 sec	0.11 sec
Avg. Task False Idle	0.12 sec	0.10 sec	0.15 sec	0.10 sec
Time-Critical Requests	1325	1330	1328	1330
Avg. Response Time	26.9 ms	19.2 ms	33.8 ms	19.2 ms
Max. Response Time	6114 ms	118 ms	6155 ms	163 ms
Time-Limited Requests	337	338	337	339
Avg. Response Time	18.9 ms	18.1 ms	21.6 ms	18.7 ms
Max. Response Time	636 ms	123 ms	4417 ms	80.0 ms
Avg. Time Limit	11.7 ms	10.5 ms	15.2 ms	11.6 ms
% Satisfied in Time	29.6 %	28.7 %	29.7 %	28.3 %
Time-Noncritical Reqs.	3325	3310	3325	3310
Avg. Response Time	308 ms	332 ms	498 ms	553 ms
Max. Response Time	6477 ms	6664 ms	6145 ms	6621 ms
Avg. Service Time	14.3 ms	14.5 ms	15.9 ms	16.0 ms
% Disk Buffer Hits	0.8 %	0.7 %	0.8 %	0.7 %
Avg. Seek Distance	128 cyls	144 cyls	172 cyls	184 cyls
Avg. Seek Time	4.9 ms	5.1 ms	5.8 ms	6.0 ms

Table 7.8: Criticality-Based Scheduling of the *synrgen4* Workload.

7.3.1 Individual Task Workloads

As described in section 5.2, the individual task workloads (figures 7.7–7.9 and tables 7.3–7.5) consist mainly of a single user task. The elapsed times shown in graph (a) of each figure represent the time from when the process executing the task begins to when it exits (i.e., completes).[4] The elapsed time is broken into four regions: false idle time, false computation time, CPU time used by the task-executing process, and other CPU time[5]. Graph (b) in each figure shows a subset of the elapsed time, excluding the CPU time used by the task-executing process. This CPU time is independent of storage subsystem performance and is not affected by changes to the disk request scheduler. The purpose of graph (b) is therefore to highlight those portions of the elapsed time that are affected by storage subsystem performance. In each figure, graphs (c) and (d) show storage subsystem performance across all I/O requests, including those initiated in the background after the task-executing process completes.

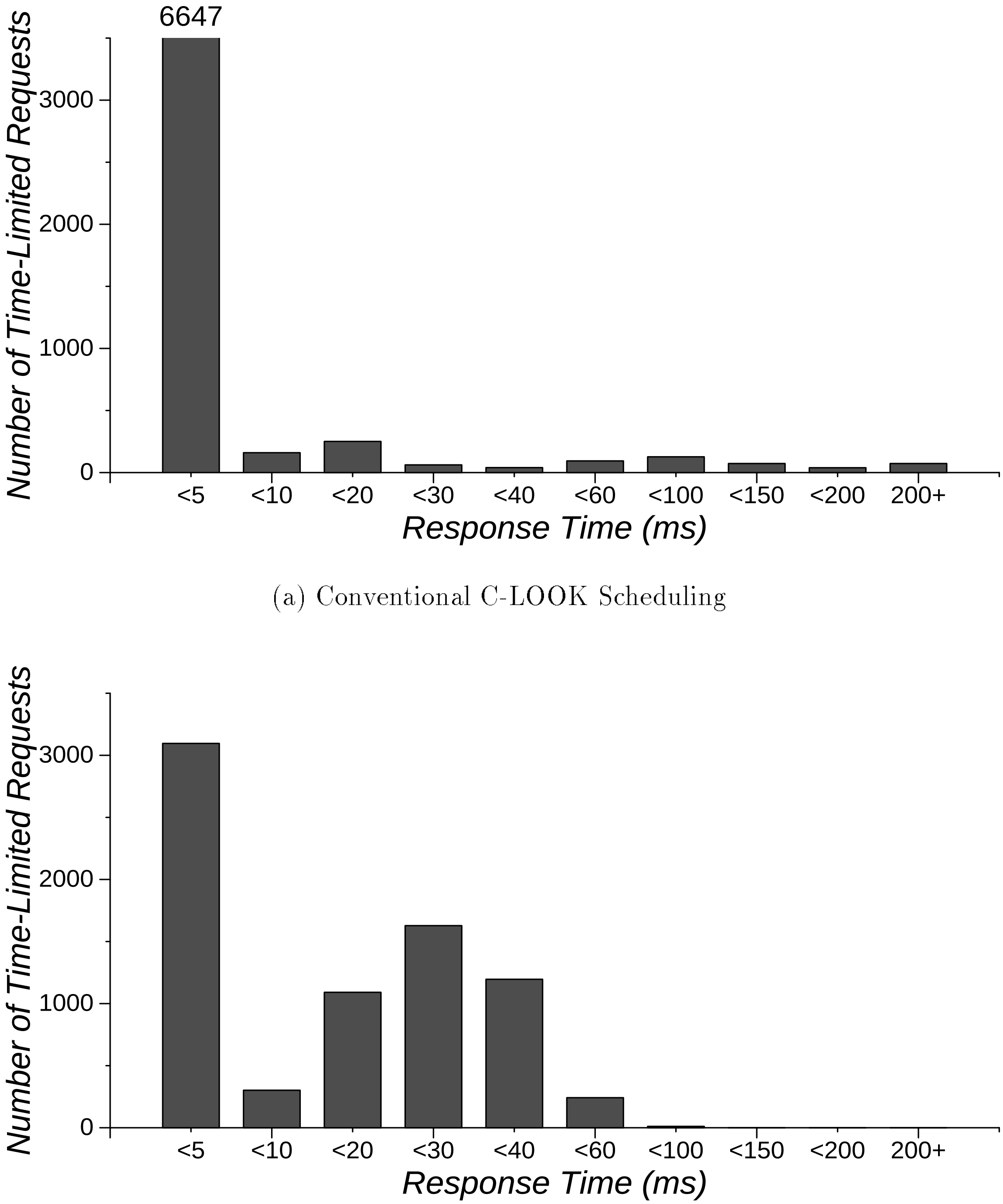
compress

Figure 7.7 and table 7.3 compare the different algorithms with the *compress* workload. The CPU time used by the task-executing process accounts for almost 83% of the elapsed time, leaving little room for improvement. Using request criticality information with C-LOOK scheduling shaves only 0.1% off of the total elapsed time. The improvement comes from a 0.5% reduction in false idle time and a 1.6% reduction in other CPU time. The former, although disappointing in scope, is an expected effect of the algorithm change. The latter is a secondary effect of the shorter elapsed time. Most of the other CPU time consists of interrupt service routines and system daemon processes. Reducing the elapsed time for the task in turn reduces the amount of background system processing during the task's lifetime. As noted above, the criticality-based FCFS algorithm results in system performance that is very close to that for the criticality-based C-LOOK. With the conventional algorithms, on the other hand, FCFS results in a 2.8% longer elapsed time than C-LOOK.

Using criticality information in the disk scheduler improves performance by reducing the amount of time that processes spend waiting for time-critical and time-limited I/O requests. It is very interesting to note, therefore, that the average response time for time-limited requests actually increases from 10.4 milliseconds to 15.3 milliseconds for C-LOOK. As a result, the percentage of time limits that are satisfied falls from 96.0% to 76.4%. The response time densities for time-limited requests shown in Figure 7.13 help to explain this phenomenon. There are two important differences between the densities:

[4] 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 (see section 5.3).

[5] The other CPU usage 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.



(a) Conventional C-LOOK Scheduling

(b) Criticality-Based C-LOOK Scheduling

Figure 7.13: Response Time Densities for Time-Limited Requests (*compress*).

- A significant tail is evident in the response time density for the conventional C-LOOK algorithm, culminating in a maximum response time of 643 ms. It is the requests with very long response times that cause large periods of false idle time. On the other hand, the criticality-based C-LOOK exhibits no tail and has a maximum response time of 93 ms.

- The conventional C-LOOK algorithm handles almost 88% of the time-limited requests in less than 5 ms. These low response times result from hits in the disk's on-board prefetching cache. The percentage of very low response times falls to 41% for criticality-based C-LOOK. However, the nature of time-limited requests is such that servicing them in zero time (or in less than 5 ms) is no better than completing them just before their time limits expire. So, while the number of time limits missed does increase, they are not missed by much, and this negative effect is not significant enough to offset the benefit of removing the very long response times.

The reduction in the number of on-board disk cache hits is a direct result of giving priority to the more critical requests, which for this workload tend to be sequential reads. The time-limited read I/O requests are generated at fairly regular intervals. The file system prefetches file block N+1 when the process accesses block N. Therefore, the number of pending time-limited requests ranges between 0 and 2. The time-noncritical requests in this workload, mainly background flushes of dirty file blocks, arrive in bursts when the syncer daemon awakens. At some point after a burst arrives, there will be 0 pending read requests and the disk scheduler (either variant) will initiate a time-noncritical write request.

At this point, the two algorithms part company. The conventional C-LOOK algorithm continues to service the time-noncritical requests independent of time-limited read request arrivals, because the time-noncritical requests generally exhibit spatial locality and will therefore incur smaller seek delays. So, the time-limited requests (and therefore the process) wait for the entire group of writes to be serviced and are then serviced in sequence as they arrive. On the other hand, criticality-based C-LOOK services new time-limited read requests before time-noncritical requests. Time-noncritical requests are only serviced when there are no pending time-limited requests. As a result, the time-limited sequential reads are interleaved with time-noncritical writes, reducing the effectiveness of the on-board disk cache and more frequently moving the disk head between multiple regions of locality. This distinction in behavior accounts for both differences in the response time densities.

Finally, as expected, giving preference to system needs rather than mechanical delay reductions reduces storage subsystem performance significantly. The average request service time almost doubles, and the average response time increases by more than a factor of 18. While the criticality-based FCFS algorithm provides roughly equivalent system performance to the criticality-based C-LOOK, it does not match C-LOOK's storage subsystem performance. With a heavy workload, FCFS scheduling will bottleneck performance before C-LOOK does.

uncompress

Figure 7.8 and table 7.4 compare the different algorithms with the *uncompress* workload. Unlike the *compress* workload, the CPU time used by the task-executing process accounts for only 61% of the elapsed time in *uncompress*. Using request criticality information with C-LOOK scheduling reduces the elapsed time by 30%. Most of the improvement comes from a 90% reduction in false idle time. The remainder comes from a 45% reduction in other CPU usage. The criticality-based FCFS algorithm results in system performance that is very close to that of criticality-based C-LOOK. With the conventional algorithms, on the other hand, FCFS results in a 16% longer elapsed time than C-LOOK.

The behavior of the response time densities for time-limited requests is similar to that described above for the *compress* workload. However, the increase in time-noncritical I/O activity relative to time-limited activity extends the very long response times, causing the high average value. Eliminating these long response times more than compensates for the effect of increasing the service times, causing the reduction in average response time (from 42.5 ms to 19.3 ms).

While system performance increases, storage subsystem performance drops dramatically. The average service time increases by 51% while the average response time increases by more than an order of magnitude. Although the increased service times account for some of the response time growth, the reduction in false idle time accounts for most of it. Because the task-executing process spends less time waiting for I/O requests, it progresses more quickly and therefore generates new I/O requests more quickly. The criticality-based FCFS scheduler also suffers a large drop in storage subsystem performance and will bottleneck system performance more quickly than criticality-based C-LOOK.

copytree

Figure 7.9 and table 7.5 compare the different algorithms with the *copytree* workload. Unlike *compress* and *uncompress*, the CPU time used by the task-executing process accounts for only 19% of the elapsed time in *copytree*. Over 68% consists of false idle time. Using criticality information with C-LOOK scheduling reduces the elapsed time by 21%. Most of the improvement comes from a 25% reduction in false idle time. The remainder comes from a 31% reduction in other CPU time during the task's lifetime. The criticality-based FCFS algorithm results in system performance very close to that for criticality-based C-LOOK. With the conventional algorithms, on the other hand, FCFS results in a 14% longer elapsed time.

The basic response time trends described earlier, both for time-limited requests and across all requests, hold for this workload as well. The average service time increases by only 6.4% when criticality information is utilized, but the reduced elapsed time increases the average response time by a factor of 40 for C-LOOK scheduling. The criticality-based FCFS scheduler also suffers a large drop in storage subsystem performance and will bottleneck system performance more quickly than criticality-based C-LOOK.

7.3.2 SynRGen Workloads

The SynRGen workloads consist of processes (one per simulated user) that execute tasks interspersed with user think times to emulate the behavior of software developers. The elapsed times shown in graph (a) of each figure represent the average time from when one of these tasks begins to when it completes, corresponding to the system response time for a user command. The idle time between tasks is part of the workload, so the caveat of the individual task workloads, relating to the need for idle time to flush dirty blocks, is satisfied. The elapsed time is broken into four regions: false idle time, false computation time, CPU time (both for the particular task-executing process and otherwise) and task sleep time. Graph (b) in each figure shows a subset of the elapsed time, excluding the task sleep time. The computation times that contribute to the task sleep time are configured to match measurements from older machines (DEC DS-5000/200). As a result, the true system performance improvement falls somewhere between the values indicated by graphs (a) and (b). Finally, graphs (c) and (d) in each figure show storage subsystem performance across all I/O requests.

Figures 7.10–7.12 and tables 7.6–7.8 compare the scheduling algorithms using SynRGen workloads with different numbers of simulated users. The improvement to overall system performance achieved with the use of criticality-based scheduling increases with the number of users and comes entirely from reductions in false idle time. With 16 users, the reduction in elapsed times is between 4% and 21%, depending on the true value of the task sleep time. That is, the improvement shown in graph (a) is 4% and that shown in graph (b) is 21%. The reduction is between 1.5% and 10% with 8 users and between 0.6% and 4.7% with 4 users. As noted previously, the criticality-based FCFS algorithm results in system performance that is very similar to that for the criticality-based C-LOOK.

One might expect that increasing the number of users would decrease the amount of false idle time, rather than increase it (as observed), by allowing the additional processes to utilize the CPU while one process is blocked waiting for I/O. However, the users spend most of their time “thinking” and the CPU utilization is low, ranging from 7% with 4 users to 35% with 16 users. The disk utilization is also low on the average, ranging 8% with 4 users to 40% with 16 users. However, the I/O requests are generated in a very bursty manner, resulting in significant queueing and therefore ample opportunity for disk request scheduling. More importantly, the use of criticality information improves performance only in those cases when both time-noncritical requests and more critical requests are waiting for service concurrently. Increasing the number of users tends to increase the frequency with which this occurs without enough of an increase in CPU work to compensate. With more users and/or more realistic emulation of computation times (something other than *sleep*), the opposite effect might be observed.

Finally, as expected, giving preference to system needs rather than mechanical delay reductions reduces storage subsystem performance. Also, reducing false idle time allows processes to progress more quickly, thereby generating I/O requests more quickly. The subsystem performance drop tends to increase with the number of users, as did the system performance improvement. With 16 users and C-LOOK scheduling, the average service time increases by 6.5% and the average response time increases by 27%. The corresponding increases are 2.8% and 33% with 8 users and 1.4% and 6.0% with 4 users. Although the subsystem performance penalties for FCFS are smaller, the absolute values are still inferior to those for C-LOOK.

7.4 Aging of Time-Noncritical Requests

Giving priority to time-critical and time-limited requests can lead to starvation of time-noncritical requests. Such starvation can cause two significant problems, relating to the fact that most time-noncritical requests are background flushes of dirty file cache blocks. First, excessively long response times may violate system guarantees about data hardness (i.e., when dirty data blocks are written stable storage). To support such guarantees, the disk scheduler must prevent individual request response times from exceeding predetermined limits. [Ganger93] evaluates a modification to the algorithms described above, wherein time-noncritical requests are moved into the high-priority queue after waiting for a certain period of time (e.g., 15 seconds). This approach was found to effectively limit the maximum response times. However, this approach also costs a considerable fraction (e.g., 30%) of the performance improvement from using criticality information. Better approaches can be devised.

A second potential problem occurs because flushed cache blocks cannot be re-used until the write I/O requests complete. If the write working set exceeds the file cache, then the system will be limited by the throughput of the storage subsystem. In this case, scheduling based on request criticality can be detrimental because it tends to reduce storage subsystem performance. [Bennett94], a re-evaluation of the work in [Ganger93], found that criticality-based disk scheduling consistently hurts performance on a memory-starved machine. More balanced systems will suffer from this problem only when the short-term write working set exceeds the file cache capacity. Measurements of real systems indicate that this occurs infrequently [Ruemmler93a]. Further, most modern operating systems integrate the file block cache with the virtual memory system [Gingell87, Moran87], allowing it to utilize much of the available memory capacity in such situations.

7.5 Summary

This chapter demonstrates that storage subsystem performance metrics do not, in general, correlate with overall system performance metrics. Criticality-based disk scheduling is shown to improve overall system performance by up to 30% while reducing storage subsystem performance dramatically (by more than 2 orders of magnitude in some cases). As a result, 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 design point [Geist94]. One interesting effect of criticality-based scheduling algorithms is that positioning-related scheduling decisions, which have long been viewed as critical to performance, become much less important to overall system performance in sub-saturation workloads.

CHAPTER 8

Conclusions and Future Work

8.1 Conclusions

This dissertation demonstrates that 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 over-estimate performance changes and often allow unrealistic concurrency in the workload. 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% are observed. The amount of error for a given subsystem change depends upon request criticality, workload intensity and the magnitude of the performance improvement. Closed subsystem models assume unqualified feedback, generating a new request to replace each completed request. As a result, closed subsystem models tend to under-estimate 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 storage subsystem metrics, such as the mean I/O 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 information by the disk scheduler is shown to reduce elapsed times for user tasks by up to 30%, whereas the mean response time increases 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% in all comparisons).

8.2 Directions for Future Research

Given the problems associated with conventional methodology, the obvious next step is to re-evaluate previous storage subsystem research using the methodology proposed in this dissertation. Section 3.3.4 describes several research topics that have been explored using the flawed conventional methodology. A good system-oriented methodology may identify erroneous conclusions in these studies.

The work described in this dissertation can be expanded in several directions:

Chapter 7 introduces criticality-based disk scheduling and evaluates some simple algorithms. More sophisticated algorithms can be devised by considering several factors:[1] (1) Scheduling requests based on both seek times and rotational latencies is generally superior to scheduling based only on seek times [Seltzer90, Jacobson91]. (2) The algorithms explored in this dissertation do not exploit the difference between time-critical and time-limited requests. Proper exploitation of this difference will require accurate estimation of time limits. (3) The scheduler should temper criticality-related goals with subsystem throughput goals to avoid excessive starvation of time-noncritical requests and because request criticality is an inherently short-sighted classification. In the long run, it may be better to ignore request criticality in some cases. (4) State-of-the-art disk drives are equipped with on-board request queues, complicating the use of criticality information.

One important component of a good performance evaluation infrastructure is a large and varied set of benchmark workloads. In the case of system-level simulation modeling, this consists largely of system-level traces. Because of the popularity of the trace-driven storage subsystem simulation, many companies include storage request trace acquisition instrumentation in their operating system software [Rama92, Ruemmler93, Treiber94]. Instrumentation for collecting system-level traces, as described in section A.2.2, should also be included. With such instrumentation, large libraries of system-level traces can be collected from real user environments.

The instrumentation described in section A.2.2 limits the length of system-level traces to the capacity of the 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 of the validity of this approach would be needed, but it offers one method of acquiring very long system-level traces.

Finally, I believe that computer system designers need to focus much more attention on the I/O subsystem (storage, network and user interface). The common view holds that computer systems consist of three main components: CPU, main memory and I/O. This is also the commonly perceived order of importance. However, technology and application trends are conspiring to dramatically increase the importance of I/O activity. In a growing number of environments, computer system performance is determined largely by I/O performance. This dictates a change in paradigm with a much greater focus on data movement.

[1] Some of these factors are explored in [Worthington95a].

APPENDICES

APPENDIX A
Detailed Description of the Simulation Infrastructure

This appendix describes the simulation infrastructure used for performance evaluation of storage subsystem designs. The simulator itself is described, including the general simulation environment, the storage subsystem components and the host system components. The simulator can be configured to model a wide range of organizations at a variety of levels of detail. As a result, many different input workloads can be used to exercise it. The supported input formats and the current library of traces are described.

A.1 The Simulator

A.1.1 Simulation Environment

The simulator is written in C and requires no special system software. General simulation support (e.g., parameter checking and simulated time management) has been incorporated both to simplify the implementation of component modules and to improve simulation efficiency. A complete simulation run consists of the six phases described below. General simulation support related to each phase is also described.

Phase 1: Read and check parameters

A simulation begins by reading configuration parameters and checking that the values are acceptable. The simulation environment provides general functions for reading a parameter value and checking it against upper and lower bounds. More sophisticated parameter checking is also performed when necessary. The simulator is very general, allowing the configuration file a great deal of latitude in selecting and organizing components. Phase 1 proceeds as follows:

1. Read the main command line parameters: (1) the file containing the configuration parameters (all steps of phase 1, other than the first and the last, read information from the configuration file), (2) the file that should contain the output results of the simulation run, (3) the format of the input trace (see section A.2), (4) the file containing the input trace, (5) the type of input workload (see section A.2), which directly determines the type of simulation (standalone storage subsystem model or system-level model).

2. Read general simulation control parameters,[1] including the endian-ness (big or little) of the machine on which the simulation will run, seeds for the random number generator (which is rarely utilized for anything other than initial state generation), the warm-up period length, indications of which result statistics should be produced, time scaling factors, and information to assist with the translation of input traces.

3. Read the parameters that define the quantity and characteristics of the components (described below) to be simulated in the particular run. Most of the associated simulation structures are allocated at this point.

4. Read the parameters that define the physical connections between the simulated devices. For example, disk 1 connects to bus 1 and disk 2 connects to bus 2.

5. Read the parameters that define any logical combinations of simulated devices. For example, data are striped across disks 1, 2 and 3, which appear to the host as a single logical device.

6. Read parameters defining the characteristics of the host system, if the simulation run functions as a system-level model. As indicated earlier, a command line parameter determines whether a given simulation run behaves as a storage subsystem model or a system-level model.

7. Read parameter override information from the command line. Commonly modified parameters can be overridden from the command line, obviating the need to have separate parameter files for each distinct simulation run.

The separation of component definitions and their interconnections (both physical and logical) is an important aspect of a large simulator [Satya86]. It greatly reduces the effort required to develop and integrate new components, as well as the effort required to understand and modify the existing components.

Phase 2: Initialize simulation state

After all parameters have been read and checked, the internal state is prepared for the ensuing simulation run. This includes initializing the component structures, establishing the connections between components (and checking that the specified connections result in a legal setup) and initializing the various statistics.

Phase 3: Warm-up period

After initialization is complete, the simulation proper begins. The warm-up period proceeds in exactly the same manner as normal simulation execution (described below). When the warm-up period ends, all statistics in the simulator are reset to their original values. The rest of the simulation state remains unchanged by this action. The warm-up period can be specified in terms of simulation time or storage I/O request count.

[1] Each parameter is checked immediately after it is read (i.e., before moving to the next value).

Phase 4: Simulation proper

The simulator maintains an internal list of upcoming events, ordered by the simulated time at which they are to occur. The **simulated time** models the wall clock in the simulator's world and is maintained as a monotonically increasing, double-precision (64-bit) floating point number.[2] The simulator proceeds by removing the first event in time order from the internal event list and servicing it (i.e., handing it off to the appropriate component module for service). The simulated time is updated to reflect the most recent event. When servicing an event, a component module will often generate new events that are then added to the internal list. The simulator continues until one of several stopping conditions is reached. In most cases, the simulator is configured to stop when the last entry of a trace file is utilized. This avoids measuring the warm-down period during which the activity still in the simulator is completed.

The simulator executes as one monolithic process with a single thread of control. This differs from simulation environments that consist of communicating threads of control (e.g., [Ruemmler94]). When using a separate thread of control for each distinct sequence of activity, one can rely on the stack to maintain context information while a thread waits for simulation time to advance. With a single thread, context information must explicitly be kept separate from the stack for each sequence of activity in each module. Generally speaking, the single thread approach is more efficient (e.g., less time spent switching between threads for communication purposes) and more portable (e.g., no special multi-threading support required). However, the thread software abstraction is convenient for module development.

After the first phase of simulation (parameter read-in), all memory allocation and de-allocation is internally managed (except stack space, of course). The simulator has been carefully designed to use a general 64-byte (16 32-bit integers) event structure for all dynamic memory needs. The simulator maintains a list of free event structures. When one is needed, it is taken from this list. If the list is empty, a page of memory is allocated and partitioned into 64-byte structures that are placed in the free list. De-allocating a structure consists of simply adding it to the free list. This approach has been very successful at reducing both simulation times and memory utilization. The system support for dynamic memory management in C (e.g., *malloc* and *free*) requires considerable CPU time and suffer from fragmentation.

As it executes, the simulator performs a good deal of self-checking. That is, the various modules are constantly verifying that their internal state remains consistent and that the activities being performed are reasonable. This self-checking has helped to identify many obscure bugs that might have otherwise gone unnoticed. When simulation efficiency is critical, many of these self-checks can be bypassed and/or excluded by the compiler.

[2] Maintaining the time in floating point can be dangerous in a simulation setting because the granularity of each value decreases in resolution as time progresses. However, 64-bit floating point numbers provide sub-nanosecond resolution over an 8-week period of time. This more than satisfies the requirements of this simulator. A 64-bit integer would be preferable. However, the C compilers present on local systems do not provide a 64-bit integer support. Constructing this data type and the corresponding operators would be an unpleasant and, in this case, unnecessary task.

Phase 5: Clean-up statistics

After the simulation proper completes, the statistics are finalized and set to a state that is conducive to printing easily. For example, the amount of idle time is measured by keeping track of when idle periods begin and end. The length of each idle period is added to a running count. If the simulation ends in an idle period, the length of the final period will be added to the running count in this phase.

Phase 6: Output results

The final phase of a simulation run writes the results to a file (optionally *stdout*) in ASCII format. A unique identifier precedes each individual datum (a distribution is treated as one datum) to ease the burden of external tools that extract particular values (e.g., for a user or a graphing utility). The simulation environment provides a general statistics module, providing the average, standard deviation, and density/distribution for a selected variable. The granularity of the density function can be configured as desired. The module provides print functions.

A.1.2 Storage Subsystem Components

The simulator contains modules for most secondary storage components of interest. Some of the major functions (e.g., request queueing/scheduling and disk block caching) that can be present in several different components (e.g., operating system software, intermediate controller, disk drive) have been implemented as separate modules that are linked into components as desired. The supported physical interconnections and logical device combinations are also described.

A.1.2.1 Component Modules

Device Driver

In the simulator, the device driver interfaces with the "rest of the system," which is either the host system portion of the simulator or the workload generator (both are described below). Storage I/O requests "arrive" via a single interface function and are placed in the appropriate per-device queue. When appropriate, given the subsystem's state and the particular configuration being simulated, accesses are scheduled to the simulated storage subsystem hardware components. The routines for handling interrupts generated by storage subsystem components are also located in this module. The device driver module can be configured to ignore the other storage subsystem components and model storage subsystem behavior in a trivialized manner, such as using access (or response) times provided with each request or a constant per-request access time. This functionality is useful for debugging host system components and for quickly extracting statistical information about a trace.

The device driver module is configured by three sets of parameters:

1. a value that indicates what form (if any) of model simplification should be performed, together with the constant access time (if appropriate).

2. values that configure the disk request scheduler located in the device driver. This includes the values described below for the general disk request scheduler and an indication of whether or not queues located in other storage subsystem components should be used if they are present (i.e., whether the device driver will allow more than one request per device to be outstanding at a time).

3. a scaling parameter that determines the CPU overheads for the various device driver functions (e.g., interrupt handling routines, request scheduling and initiation, new request arrival processing). The simulator currently uses internal constants for these values, applying the single scaling parameter to all of the overheads. For example, a scale factor of 1.0 uses the internal constants and a scale factor of 0.0 models zero-latency CPU activity.

Controllers and Adapters

The controllers and adapters share a common module that represents all intermediate entities between device drivers and storage devices (disk drives in most storage subsystems). This module currently supports three types: (1) a bus adapter that simply passes all data and control information directly to the next entity on the I/O path without processing it, (2) a simple controller that deals with predetermined sequences of control signals but requires considerable assistance (usually provided by the device driver) to manage its state. This submodule roughly emulates the NCR 53C700-based controller used in the experimental system described below, (3) an intelligent controller that optionally includes request queueing/scheduling and/or disk block caching functions. An intelligent controller manages the storage devices connected to it and presents an interface to the rest of the storage subsystem that is very similar to the interface that the device driver presents to the rest of the system.

All instances of the controller module are configured with three important parameters:

1. the type of controller (one of the three described above).

2. a value that determines command processing overheads. The simple bus adapter uses this value as a constant per-command CPU overhead. The other controller types use this value as a scale factor applied to the hard-wired internal overheads. For example, a scale factor of 1.0 uses the internal constants and a scale factor of 0.0 models zero-latency CPU activity.

3. the time required to transfer a single 512-byte sector into or out of the controller/adapter. The simulated transfer speed is the maximum of this value and the corresponding values for the other components involved in the communication, including the bus(es). The particular submodule implementation determines whether the controller acts as a simple speed-matching buffer for data transfers or as a store-and-forward buffer.

For instances of the intelligent controller submodule, there are additional parameters, falling into two sets:

1. values that configure the request queue/scheduler, including the values described below for the generic disk request scheduler, the maximum number of requests that the

controller will schedule concurrently to each storage device (if the device supports request queueing), and the maximum number of requests (total across all managed storage devices) that can be outstanding to the controller at a time. During the initialization phase, components above the controller (e.g., device drivers) "ask" for the latter value and thereafter do not compromise it.

2. values that configure the disk block cache. These values are described below for the generic disk block cache.

The simple bus adapter submodule is trivial, simply forwarding any control messages to the next entity along the I/O path (toward the host or toward the storage devices, as appropriate). Data transfers also pass thru the bus adapter, which acts as a simple speed-matching buffer. The simple controller submodule, which emulates the behavior of a controller based on the NCR 53C700 [NCR90], immediately forwards incoming storage I/O requests down the I/O path. The controller can deal with simple sequences of control activity on its own but requires assistance with major state changes, such as SCSI device disconnects and reconnects [SCSI93]. It gets this assistance by sending an interrupt up the I/O path. An intelligent entity (i.e., the device driver or an intelligent controller) manages the controller's state change in the corresponding interrupt handling routine. Data transfers pass thru the simple controller, as with the bus adapter.

The intelligent controller submodule, together with the storage devices that it manages, behaves like a self-contained storage subsystem. Incoming storage I/O requests enter the cache module. If no cache is being simulated, these requests pass immediately to the corresponding per-device request queue. Otherwise, space for the data associated with the request is allocated and/or locked. For read hits, the data are immediately transferred from the cache and completion is then reported (i.e., a completion interrupt is sent up the I/O path). For writes, the data are immediately DMA'd from the source memory to the cache. This submodule uses the cache as a store-and-forward buffer. All data are staged through the cache, transferring into and out of the controller in two distinct steps. Completion for write requests may be reported at this point, depending on the simulated cache's configuration. The cache module generates accesses for writes and read misses and places them in the appropriate per-device queues. Accesses are scheduled from these queues to the corresponding devices according to the simulated scheduler's configuration. An intelligent controller manages all interaction with the storage devices connected to it, interrupting entities above it in the I/O path only to signal request completions.

Buses

All interconnections between storage subsystem entities (e.g., device drivers, controllers/ adapters and disk drives) share a common module. This module is architected so that communicating entities need not know the characteristics of the entities they communicate with or of the bus itself. The module supports two bus types: an ownership-based bus and an interleaved-transfer bus. With the former, an entity arbitrates for bus ownership and keeps it for as long as necessary to complete a communication sequence with another entity. An example of this is a SCSI bus [SCSI93]. With the latter bus type, separate communication sequences involving distinct entities are interleaved, such that each ongoing sequence receives an equal fraction of the available bus bandwidth. Commonly, system backplanes function in this way.

An instance of the bus module is configured with five parameters:

1. the bus type (one of the two described above).
2. the arbitration policy, which is used only for ownership-based buses. The module currently supports **First-Come-First-Served** arbitration, **Slot-Priority** arbitration (as used with SCSI buses), and a pseudo-**Round-Robin** arbitration policy.
3. the time required to perform arbitration.
4. the time required to transfer a single 512-byte sector from a bus controlling entity to another entity on the bus. The simulated transfer delay is the maximum of this value and the corresponding values for the other components involved in the communication, including other bus(es).
5. the time required to transfer a single 512-byte sector to a bus controlling entity from another entity on the bus.

While the specifics of inter-component communication are described below, the bus module interacts with other storage subsystem components in four important ways: (1) A component seeks bus ownership to communicate with another by calling into the bus module and waiting (in simulated time) for it to respond positively. Interleaved-transfer buses, of course, do so immediately after the arbitration delay and reduce the bandwidth concurrently used by other communicating components. Ownership-based buses wait for the bus to be free and then perform arbitration, responding positively to the winner after the arbitration delay. Components call into the bus module when they finish using the bus, freeing the resource for others to use. (2) Transmission delays for control information and processing delays that occur while the bus is held can be performed by calling into the bus module. After the appropriate delay elapses in simulated time, the bus module will call back into the appropriate component module. (3) Control information is passed from one entity to another via the bus module. (4) The bus module contributes to determining the speed of data transfers, although other components manage the transfer.

Disk Drives

Disk drives remain the de facto standard in secondary storage. These mechanical storage devices have grown very complex over the years, both because of advances in mechanical engineering and because of increased use of on-board logic. The latter has allowed disk firmware developers to employ various algorithms and policies to hide the specifics of media access, to dynamically handle media corruption, to hide/reduce media access latencies and to increase on-media storage densities. The disk drive module accurately models most aspects of modern disk drives, including zoned recording, spare regions, defect slipping and reallocation, disk buffers and caches, various prefetch algorithms, fast write, bus delays, and command processing overheads. Configuring an instance of the disk drive module requires over 100 different parameters. Because of its complexity, a description of the disk drive module will be deferred to appendix B.

Request Queues/Schedulers

Because request queues and the corresponding schedulers can be present in several different storage subsystem components (e.g., device drivers, intelligent controllers and disk drives), request queue/scheduler functionality is implemented as a separate module that is incorporated into various components as appropriate. New requests are referred to the queue module by queue-containing components. When such a component is ready to initiate an access, it calls the queue module, which selects (i.e., schedules) one of the pending accesses according to the configured policies. When an access completes, the component informs the queue module. In response, the queue module returns a list of requests that are now complete. This list may contain multiple requests because some scheduling policies combine sequential requests into a single larger storage access. The queue module collects a variety of useful statistics (e.g., response times, service times, inter-arrival times, idle times, request sizes and queue lengths), obviating the need to replicate such collection at each component.

An instance of the queue module is configured with a dozen parameters that fall into seven groups:

1. the base scheduling algorithm. The simulator currently supports 18 different disk scheduling algorithms, including **First-Come-First-Served**, LBN-based and cylinder-based versions of common seek-reducing algorithms (**LOOK**, **C-LOOK**, **Shortest-Seek-Time-First** and **V-SCAN(R)**), and a variety of **Shortest-Positioning-Time-First** algorithms (combinations of cache-aware, aged, weighted and transfer-aware). Most of the supported scheduling algorithms are described and evaluated in [Worthington94].

2. the information used to translate logical block numbers (LBNs) to cylinder numbers. This translation is used only for the cylinder-based seek-reducing algorithms. Several options are supported, including a constant average number of sectors per cylinder, accurate per-zone information (first cylinder of zone and number of sectors per cylinder), accurate information regarding the first LBN of each cylinder, per-zone information and format-time slipping but not defect remapping, and full mapping information (as used by the disk itself).

3. read and write request initiation delays, for use in the **Shortest-Positioning-Time-First** (SPTF) algorithms. These algorithms must estimate what the rotational offset of the read/write head will be when re-positioning begins. Together with the current offset and rotation speed, these values are used to calculate the required estimate.

4. the age factor for the aged and weighted versions of the SPTF algorithm. These algorithms are described in [Seltzer90, Jacobson91, Worthington94].

5. values that configure algorithm enhancements for exploiting request sequentiality. The enhancements fall into two categories, concatenation and sequencing, that can be applied selectively to reads and/or writes. A parameter value provides the four bits in this cross-product. Request concatenation consists of combining sequential requests into a single larger access. A second parameter specifies the maximum allowed size of this larger access. Request sequencing consists of scheduling pending sequential requests

in ascending order. A third parameter optionally allows some amount of “give” in the definition of sequential. Two requests are treated as sequential if the second starts within this number of sectors after the first. The one exception is write concatenation, which can only be performed with truly sequential requests.

6. values for configuring a separate timeout queue. Requests that wait in the queue for longer than some pre-determined amount of time are placed in the second queue. Requests are scheduled from the timeout queue before the base queue is examined. A parameter value specifies whether or not the timeout queue is utilized. Two other parameters specify the timeout time and the scheduling algorithm used for the timeout queue, which may differ from that used with the base queue.

7. values for configuring a separate priority queue. Time-critical and time-limited requests can be given non-preemptive priority over time-noncritical requests by placing them in a separate queue. Requests are scheduled from the priority queue before the base queue is examined. A parameter value specifies whether or not the priority queue is utilized. A second parameter specifies the scheduling algorithm used for the priority queue, which may differ from the base queue. Chapter 7 uses this functionality to examine criticality-based disk scheduling.

Disk Block Caches

Because disk block caches can be present in multiple components (e.g., host main memory and intelligent storage controllers), implemented disk cache functionality is implemented as a separate module that is incorporated into various components as appropriate.[3] Requests are referred to the cache module when a cache-containing component chooses to begin servicing them. The cache module deals with servicing the request and informs the component when completion can be reported. Cache containing component modules must provide several call-back functions for the cache module, including those for issuing transfer requests to move data into and out of the cache, those for telling the component to continue a previously blocked cache-accessing process (this is particularly important for the host memory cache) and those for indicating request completion. As appropriate, the cache deals with the decoupling of requests from above and the flush/fill requests that access backing store devices.

An instance of the disk cache module is configured with many parameters, falling into 11 groups:

1. the cache size (in 512-byte sectors).

2. the cache line size (in 512-byte sectors).

3. the valid and dirty bit granularities (in 512-byte sectors). This parameter determines the number of sectors covered by a single valid (dirty) bit. The sectors covered by a single bit must either all be valid or not valid (dirty or not dirty) in the cache at any point in time. As a result, the valid (dirty) bit granularity affects the flush/fill requests issued to the backing store. This parameter must evenly divide the cache line size.

[3] The disk drive module includes a separate cache/buffer submodule because of the specialized nature of common on-board disk cache management policies.

4. values defining the cache data locking strategy. Appropriate locks must be held in order to access cache contents. One parameter specifies the lock granularity (in 512-byte sectors), indicating the number of sectors covered by a single lock. The lock granularity must evenly divide the cache line size. A second parameter indicates whether read locks are shared or held exclusively.

5. the maximum size (in 512-byte sectors) of a request from above. During the initialization phase, the device driver requests this value. During the simulation proper, the device driver breaks excessively large requests into multiple smaller requests.

6. the replacement policy for cache lines. Several policies are currently supported, including **First-In-First-Out**, **Last-In-First-Out**, pseudo-random, **Least-Recently-Used** (LRU), and **Segmented-LRU** [Karedla94]. A second parameter specifies the number of segments for the Segmented-LRU policy.

7. a flag specifying whether a space-allocating process waits for a reclaim-time dirty line flush (i.e., when the line to be replaced is dirty) or attempts to find a clean line to reclaim.

8. the write scheme. Three options are currently supported: (1) synchronous write-thru, wherein newly written data are copied to the backing store before request completion is reported, (2) asynchronous write-thru, wherein a flush access for newly written data is generated, but not waited for, before completion is reported, (3) write-back, wherein completion is reported for write requests as soon as the new data have been moved into the cache. Individual requests can also selectively enforce the synchronous write-thru scheme, as required.

9. values defining the flushing strategy for write-back caches. One parameter defines the main approach, which must be one of two currently supported options: (1) demand-only, wherein dirty data are written to the backing store only when the space needs to be reclaimed, (2) periodic, wherein a background "process" periodically flushes some or all of the dirty cache data. The latter option is further configured by two parameters that specify the period and the number of times during the period the process awakens. The latter value also defines the fraction of the cache contents examined for dirty lines to be flushed each time the process awakens. Optionally, dirty cache data can also be flushed during idle time. A parameter specifies whether or not this is done and how long a backing store device must be idle before such flushing begins. An additional parameter determines the maximum amount of clustering (i.e., combining multiple contiguous dirty lines into a single flush request) performed.

10. values that specify the type of data prefetching. Currently, the cache module only prefetches data when it is also performing a demand fetch. Also, the prefetching does not cross cache line boundaries. So, a single parameter specifies how much of the line, other than the required portion, is fetched. One bit specifies the beginning of the line and another the rest of the line. One such parameter covers demand fetches for servicing read requests. A second such parameter covers demand fetches for write

requests that modify only a fraction of the space covered by a single valid bit (the cache module currently assumes a write allocate policy).

11. a value that specifies whether requests are serviced with separate per-line fill or flush requests. The more efficient alternative is to allocate the full set of necessary lines and issue a single multi-line backing store access.

12. a value indicating how many non-contiguous cache memory lines that contain contiguous backing store data can be combined. Such combining requires gather/scatter hardware support, which is not always available. That is, outgoing data must be gathered from non-contiguous cache locations and incoming data must be scattered among them.

Disk Array Data Organizations

Because disk array data organizations [Chen93b, Ganger94a] can be utilized in multiple components (e.g., device drivers and intelligent storage controllers), logical address mapping is implemented as a separate module (the logorg module) that is incorporated into various components as appropriate.[4] The logorg module supports most logical data organizations, which are made up of a data distribution scheme and a redundancy mechanism [Ganger94a]. Requests are referred to the logorg module when they arrive at a logorg-containing component. The logorg module translates a newly arrived logical request into the appropriate set of physical accesses (from the components point of view) and returns a subset of these. A logical request will be translated into multiple physical accesses for many logical data organizations, such as those that employ disk striping and/or replication-based redundancy. Some of these physical accesses may have to wait for others to complete, as with the write request of a read-modify-write parity update operation for parity-based redundancy [Patterson88]. When a physical access completes, a logorg-containing component calls into the logorg module, which returns a list of requests that are now complete and a list of additional physical accesses to be performed.

An instance of the logorg module is configured with many parameters, which can be described in 6 groups:

1. the set of physical disks involved. The current implementation does not allow logical organizations to use fractions of disk drives. One parameter specifies the number of disks involved and two others specify the range. Each disk is referred to by a number in the configuration file.

2. the logical addressing scheme for the logorg. The logical organization can either be addressed by a single logical device number or by addressing the individual physical devices. In either case, the addresses will be modified, as appropriate, by the logorg module. One parameter indicates which of the addressing options should be utilized. A second parameter specifies the single logical device number.

3. the data distribution scheme. The logorg module currently supports four options: (1) as is, wherein the logical address within the logical organization's capacity remains

[4] Currently, only the device driver uses this module.

unchanged, corresponding to the conventional, independent-disk storage subsystem, (2) striped, wherein logical addresses are spread among the physical storage devices by a hashing mechanism, (3) random, wherein the logorg module randomly selects a physical device for each new request, (4) ideal, wherein the logorg module selects physical devices for new requests in a round-robin fashion. The latter two options are not realistic data distribution schemes. They are used only for comparison purposes (as in [Ganger93a]). One parameter specifies the data distribution scheme (one of the four supported options) and a second parameter specifies the stripe unit size.

4. the redundancy mechanism. The logorg module currently supports five options: (1) no redundancy, (2) replication, wherein a copy of each disk's contents are maintained on one or more other disks, (3) parity disk, wherein one of the physical disks contains parity information computed from the other disks' contents, rather than data, (4) rotated parity, wherein one disk's worth of capacity is used for parity information that is interleaved among the physical disks, (5) table-based parity, wherein a complex parity-based organization, such as parity declustering [Holland92] is utilized.

5. values that configure replication-based redundancy mechanisms. One parameter specifies how many copies of each disk are maintained. This value must divide evenly into the number of physical disks. A second parameter specifies how the logorg module decides which copy is accessed for each read request. Since each copy contains the same data, a read request can be serviced by any of them. Supported options include: (1) primary copy only, (2) random, (3) round-robin, (4) shortest expected seek distance, and (5) shortest queue length.

6. values that configure parity-based redundancy mechanisms. One parameter specifies the reconstruct-write fraction. That is, the fraction of disks in a parity stripe that must be written for reconstruct-write to be used instead of read-modify-write (see [Chen93b, Ganger94a]). Another parameter specifies the parity stripe unit size, which may differ from the data stripe unit size (although one must be an even multiple of the other). A parameter specifies the parity rotation type for the rotated parity redundancy mechanism (see [Lee91]). A parameter specifies the file that contains the block layout table for table-based parity. A final parameter specifies whether the distinct writes for read-modify-write parity updates can be issued as each read completes or must wait until all of the reads complete. In the latter case, a set of spindle synchronized disks will write the data at approximately the same time.

A.1.2.2 Physical Component Interconnections

This subsection describes the supported physical component interconnections. The allowed interconnections are roughly independent of the components themselves except that a device driver must be at the "top" of a subsystem and storage devices (e.g., disk drives) must be at the "bottom." Most of the restrictions on how components can be connected are imposed by the specific routing mechanism employed by the storage subsystem simulator, which is described below. Also described below are the message types and protocols for communication among storage subsystem components.

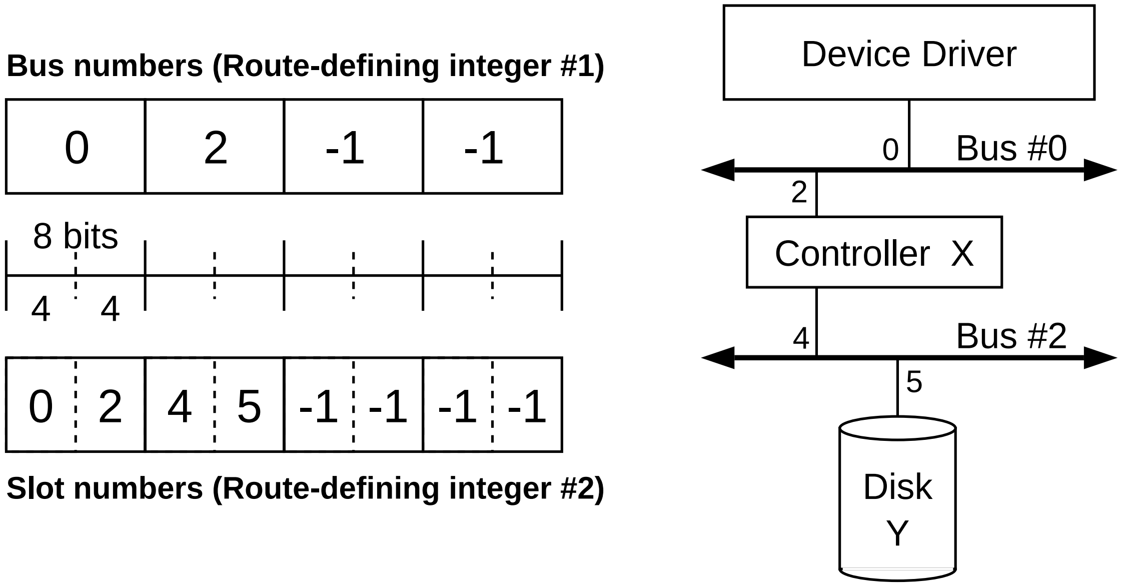

Bus numbers (Route-defining integer #1)
0
2
-1
-1
8 bits
4
4
0
2
4
5
-1
-1
-1
-1
Slot numbers (Route-defining integer #2)
Device Driver
0
Bus #0
2
Controller X
4
Bus #2
5
Disk Y

Figure A.1: Internal Storage Subsystem Message Routing. The figure shows the route-defining integer values for transferring messages to/from the device driver from/to disk Y, or any intermediate steps in said path. Note that the particular communicating components (device driver, controller X, and disk Y) are not reflected in the routing integers.

Message Routing

For each storage device (e.g., disk drive), two 32-bit integers describe the static route from the device driver to the device (see figure A.1). The first lists the buses that are traversed, using one byte to provide each bus number. Each byte in the second integer lists the two slot numbers for the communicating devices on the bus listed in the corresponding byte of the first integer, using four bits for each slot number. The upper four bits correspond to the component that is closer to the host system in the path. Each component instance knows its depth (which must be constant across all paths that contain the component) in these routing integers. The depth indicates which bytes correspond to each component in the path. So, to route a message down (up) the path, a component sends the message to the bus module with the bus number and the lower (upper) slot number taken from the byte at the next lower (higher) depth. The bus module knows which components are connected to each slot of each bus instance and can therefore route the message to the correct component module.

The configuration file specifies which components are connected to each bus, and the order of these connections defines the slot numbers associated with each. The configuration file also specifies which buses are connected to each “host-side” component. The supported hierarchy scheme currently limits the number of “host-side” components on each bus to one. These two sets of configuration file input fully defines the physical interconnection hierarchy. During the initialization phase, the storage subsystem components “feel around” to determine the per-device routes.[5] The device driver collects the per-device routes. As

[5] Currently, static per-device routes are determined at initialization time and used throughout the simulation. The component modules could be modified to deal with dynamically selecting among multiple routes, emulating dynamic path reconnection and related optimizations.

each request arrives during the simulation proper, the device driver adds the appropriate route to the internal storage I/O request structure.

The routing mechanism used in the storage subsystem simulator imposes several constraints on component connectivity:

- no more than 63 buses. The binary value '11111111' is used to represent an unused entry in the first route-defining integer.
- no more than 15 components connected to a bus. The value '1111' is used to represent an unused value in the second route-defining integer.
- no more than 1 host-side component connected to each bus.
- no more than 4 buses traversed on the route from the device driver to a storage device.

The simulator currently imposes two additional constraints:

- no more than 1 host-side bus connected to each component.
- no more than 4 device-side buses connected to each component.

Message Types and Protocols

Five types of messages are passed among storage components using the bus module. An *I/O Access* message is passed down the I/O path (i.e., towards the appropriate storage device) to initiate a storage access. An *Interrupt Arrive* message is passed up the I/O path (i.e., towards the device driver) when the disk drive attempts to initiate bus activity. The five interrupt types are (1) completion, (2) disconnect, (3) reconnect, (4) save pointers, (5) ready to transfer. The middle three interrupt types correspond to SCSI device actions [SCSI93]. The "ready to transfer" interrupt is used only when a component transfers data for a new *I/O Access* before disconnecting from the bus. An *Interrupt Completion* message is passed down the I/O path in response to an *Interrupt Arrive* message. A *Data Transfer* message is passed up the I/O path to initiate data movement. The transfer is managed by an intermediate controller, either the intelligent controller nearest the component that sends the message or the simple controller nearest the host system. A *Data Transfer Complete* message is passed down the I/O path when requested data movement completes.

Figure A.2 shows the messages that can arrive from "above" (i.e., closer to the device driver) and from "below." The expected responses to the different messages are also shown. A specific component may respond to messages (e.g., disk drives, intelligent controllers and device drivers) or may simply pass them further along the I/O path (e.g., simple adapters). Most of the messages are reactionary in nature. Only the *I/O Access* message and the "reconnect" form of the *Interrupt Arrive* message initiate new sequences of message passing. The "completion" and "disconnect" forms of the *Interrupt Completion* message end sequences of message passing.

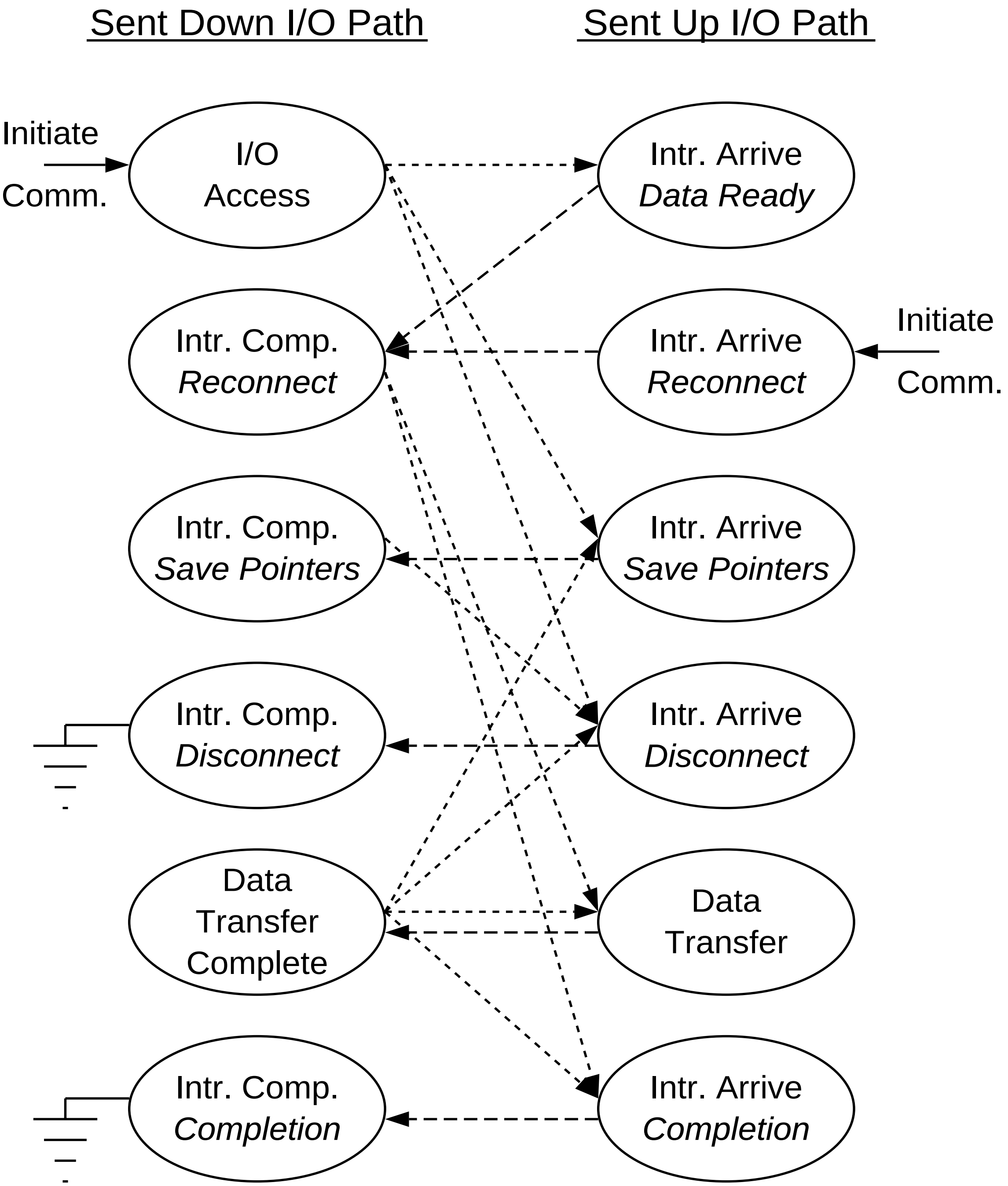


Figure A.2: Message Sequences as Exchanged by Storage Components. This represents communication activity as observed by the bus module. Each sequence is initiated either by an I/O Access message or by a Reconnect Interrupt Arrive message. Disconnect Interrupt Complete and Completion Interrupt Complete messages end sequences. The dashed and dotted lines represent the response options for messages received from below and above, respectively. Note that there is only one acceptable response to each message from below. The selected response to a message from above depends upon the initial configuration and current state of the responding component.

A.1.2.3 Logical Data Organizations

Logical data organizations, which combine the capacity of multiple storage devices (e.g., disk drives), consist of one or more instances of the disk array data organization module described above. Currently, logical device numbering is not used in message routing and all components know the routes for each physical device. This could be changed to better match the behavior of real disk arrays. All other important aspects of logical data organizations in the simulator are described above for the component module.

A.1.3 Host System Components

The simulator 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. The modules and their interactions are described.

CPUs

In the simulator, the CPUs "execute" processes and interrupt service routines, which are modeled as sequences of events separated by computation times, by decrementing each computation time (as simulated time progresses) in turn until it reaches zero, at which time the event "occurs." Each CPU structure contains pointers to the current process, the stack of current interrupts and the one event whose computation time is currently being decremented. 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 handling routine becomes the CPU's current event. Interrupt completion events remove interrupts from the stack and context switch events replace the current process.

The CPU module is configured with two parameters:

1. the number of CPUs.
2. a scale factor for computation times. With this scale factor, the effective CPU speed can be increased or decreased.

Main Memory

Main memory is not currently simulated as a distinct entity. Process-induced data movement is included in the computation times. I/O-induced DMA activity is modeled by the storage subsystem components. The emulation of memory management software, particularly disk cache behavior, is described below.

Interrupt Controller

The simulated interrupt controller is extremely simple and no configuration parameters are needed. The interrupt controller simply bundles interrupt messages in a convenient internal format and sends them to CPU #0. This module could be changed to better emulate interrupt controllers that hold low-priority interrupts until higher-priority interrupts have been handled and/or route such interrupts to other CPUs.

Operating System Software

The events that comprise processes and interrupt handlers represent important system software actions. Therefore, operating system functionality is simulated by updating the host system model's state as each of these events "occurs." In some cases, additional events will be inserted into the event sequence representing the process or interrupt handler. The system software events used in the simulator can be separated into 5 groups, each relating to one of: (1) process control, (2) system clock, (3) storage subsystem interface, (4) external interrupts, and (5) main memory management. Each of these groupings is described below. The system software aspects of the simulator are modeled after the behavior of AT&T/GIS's UNIX SVR4 MP operating system, important aspects of which are described below.

Process Control

The main policy code for the process scheduler has been extracted from the AT&T/GIS's UNIX SVR4 MP operating system and inserted directly into the simulator. Eleven events are used to emulate the process control performed in this operating system:

1. *fork*: This event causes a new child process to come into existence. The trace event includes the new process's identification number (PID). The trace event stream corresponding to the new PID is used for the new process.

2. *exec*: Many instances of the *fork* event use a variant of the UNIX *fork* command that requires the parent process to wait until the new child process either performs an *exec* command or an *exit* command. No other simulator state is modified by this event.

3. *exit*: This event ends execution for a process. All resources should have previously been freed/deallocated. This event should be the last in a particular process's event stream.

4. *sleep*: This event causes the process to stop execution temporarily. A parameter supplies a wakeup value for the process, which waits until one of two things happens: (1) another process or an interrupt handler performs a *wakeup* event with the value, or (2) another process or interrupt handler performs a *signal* event for the process.

5. *wakeup*: This event causes sleeping processes to resume execution. One parameter provides the wakeup key (mentioned above for the *sleep* event). A second parameter indicates whether all processes waiting for the key should be awakened, or only one of them.

6. *signal*: This event causes action on a specific process. Frequently, this action consists of waking the process and optionally causing it to immediately execute an *exit* event.

7. *system call*: The UNIX operating system being emulated does not switch out a process while it is actively executing system code. Therefore, it is necessary to know when a process is executing inside the kernel. A system call represents a kernel entry point. No other simulator state is modified.

8. *system call complete*: The kernel exit point corresponding to a previous *system call* event.

9. *trap*: As with system call, traps represent kernel entry points.

10. *trap complete*: The kernel exit point corresponding to a previous *trap* event.

11. *context switch*: This event causes a CPUs current process to be replaced by the process specified as a parameter.

Most of the process control events are part of the per-process trace event streams provided as simulator input. *sleep* and *wakeup* events are added when processes must wait for time-limited I/O requests, and removed when modified I/O subsystem performance obviates traced I/O waits. Also, all *context switch* events are generated internally. The process control activity is currently configured with only two parameter. The first specifies whether the system executes the idle loop in a separate idle process or within the context of the most recently executing process. The second parameter specifies whether or not processes waiting for I/O requests should preempt other executing processes when their requests complete.

System Clock

The system software receives a clock interrupt every 10 milliseconds and maintains an internal "clock" of this granularity. In the clock interrupt handler, the system increments its internal clock and updates the quanta and sleep times in the process scheduler, possibly inducing a further process control activity (e.g., context switches for processes whose quanta expire). The system performs several additional tasks on every 100th clock interrupt. For example, the process scheduler examines the run queues more thoroughly. Also, background system daemons are awakened to initiate flush requests for dirty file blocks.

The simulator generally uses the traced clock interrupt activity, including the arrival times and handler event streams. The simulator can also be configured to internally generate and handle clock interrupts appropriately. This functionality is used for the synthetic storage I/O workload generator (described in section A.2.3) and for the more complete model of memory management activity described below. In any case, interrupt handlers use one special event that processes do not use, the *interrupt complete* event.

Storage Subsystem Interface

The storage subsystem interface remains simple (as described in section 3.3). *I/O request* events issue I/O requests to the device driver. The device driver module uses several internal events, which execute in processes and interrupt handlers as described above, and to complete its work. These events are added process event streams when necessary. The device driver also dynamically constructs the storage I/O interrupt handler event streams, using its internal events as needed. The internal device driver events are:

- *Schedule Access*: This event issues a I/O access to the storage subsystem hardware.

- *Respond to Device*: This event causes the device driver to clear the interrupt for the device that initiated it. As described above with the storage subsystem components, this consists of passing an *Interrupt Complete* message down the I/O path.
- *Request Complete*: This event causes the device driver to inform the system that an I/O request has completed.

The device driver also uses *wakeup* events in request completion interrupt handlers for time-critical I/O requests.

Currently, the simulator uses the I/O requests as present in the per-process traced event streams. The memory management software component described below will skip the traced requests and generate I/O requests internally based on the modeled effects of the traced file cache accesses.

External Interrupts

One of the input files for a system-level simulation run is a file containing the traced arrival times and handler event streams for external interrupts, which are the interrupts generated by network and user interface I/O activity. These interrupts are fed into the simulator at the measured arrival times with no feedback, modeling an the input mechanism of an open system. All of the system-level traces described below and used in this dissertation were taken with the network disconnected and no interactive users. As a result, there are no external interrupts in these workloads.

Main Memory Management

A new host system module (still under development and yet to be validated) will emulate memory management software. In particular, those aspects that relate to the use of main memory to cache disk blocks in the form of file blocks and virtual memory pages. This will be an important module, as it will allow user's to examine interactions between host system caching and storage subsystem design. The cache access events are already part of the traced process event streams. The simulator currently ignores them and uses the events that are direct effects, such as *I/O Request* events for cache fills/flushes and *sleep* events for time-critical requests. When this new module is activated, this will be reversed. The simulator will use the cache events and ignore the events that are direct effects. The new module will generate the appropriate replacements for these latter events.

A.2 Input Workloads

The simulator can take three forms of input: storage I/O request traces, system-level traces and synthetic I/O workload descriptions. As indicated above, the form of input determines the type of simulation used. Storage I/O request traces drive storage subsystem simulations. System-level traces drive system-level simulations. Synthetic I/O workload descriptions utilize components of the system-level to drive storage subsystem simulations. Each of these forms of input workload are described below.

A.2.1 Storage I/O Request Traces

The simulator accepts storage I/O request traces in several formats, allowing use of the variety of traces currently available to our research group. The trace input module organization simplifies the addition of new formats. The currently supported formats include:

- The format of the disk request traces captured by the UNIX SVR4 MP device driver instrumentation described below.
- The format of the extensive disk request traces from HP-UX systems described in [Ruemmler93].
- The format of the I/O traces from commercial VMS systems described in [Rama92].
- A simple ASCII format, used mainly for debug purposes.

Internal Storage I/O Request Format

During execution, the simulator reads request information from the trace file and translates it to the internal request format. The simulator handles differences between the endianness of the machine on which the raw traces were produced and that of the machine being used for the simulation run (as indicated by a parameter file value).

The fields of the internal request format are:

- **Arrival Time**: the request's arrival time relative to the trace start time.
- **Device Number**: the internal logical device number to be accessed, before any logical-to-physical device remappings. Parameter file values indicate the mappings from trace file device numbers to internal device numbers.
- **Starting Block Number**: the internal logical starting address, in 512-byte blocks, to be accessed. Parameter file mapping values can also affect the value of this field.
- **Size**: the number of 512-byte blocks to be accessed.
- **Flags**: internal control bits that define the type of access requested. The storage subsystem currently uses only three of the bits, although others are used by system-level model. One bit indicates whether the request is a read (i.e., from the storage subsystem to host memory) or a write. Two other bits indicate whether the request is time-critical, time-limited or time-noncritical. This information can be used by the storage subsystem to improve overall system performance (see chapter 7).
- **Main Memory Page Descriptor Address**: the address of the descriptor for the source/destination main memory page(s). The storage subsystem simulator does not currently use this information, but the system-level model uses it in a manner similar to most UNIX systems. Roughly speaking, this descriptor keeps track of what the page contains, who (processes and/or I/O requests) is currently using it and how it is being used.

- **Access Time**: the measured access time for the request. As described above, the simulator can be configured to use measured access times rather than simulating the storage subsystem activity.
- **Queue Time**: the measured queue time for the request. The simulator can also be configured to use the measured response time (i.e., queue time plus access time) for each request rather than simulating the storage subsystem activity.

Trace-Driven Storage Subsystem Simulation

As described in section 3.3.3, a simulation driven by a storage I/O request trace usually emulates an open subsystem model. To accomplish this, the simulator assumes that the trace records are stored in ascending arrival time order. The simulator reads the first request from the trace during the initialization phase and adds it to the internal event queue. Handling the request arrival consists of two tasks: (1) issue the request into the storage subsystem by calling the appropriate device driver function, and (2) read the next request from the trace and add it to the internal event queue. The simulator can also be configured to use trace information to emulate a zero-think-time closed subsystem model. In this configuration, the simulator ignores arrival time information. During the initialization phase, the simulator reads N requests (where N represents the constant request population) from the trace file and adds them to the internal event queue with an arrival time of zero. The second arrival-induced task described above disappears, but request completions require additional work. After normal completion processing, the simulator reads the next request from the trace and adds it to the internal event queue with the current simulated time as its arrival time.

Storage I/O Trace Acquisition

To collect storage I/O request traces, I instrumented the device driver module of the UNIX SVR4 MP operating system that executes on the experimental system described in the next section. The instrumentation collects trace data in a dedicated kernel memory buffer and alters performance by less than 0.01%, 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 data to a file. The trace record for each request contains the device number, the starting block number, the size, the flags and three high-resolution timestamps. The timestamping mechanism (described in appendix C) combines a high-resolution (approximately 840 nanoseconds) diagnostic counter and the low-resolution (approximately 10 milliseconds) operating system "clock." The three timestamps provide the time that the request arrived at the device driver, the time that the corresponding access was scheduled to the storage subsystem hardware and the time that request completion was signaled. The difference between the first two timestamps is the queue time. The difference between the second and third timestamps is the access time. By assuming that the first request in the trace arrives at time zero, the simulator's arrival time field (see above) for a request is the difference between the first timestamp of that request and the first timestamp of the first request in the trace.

Trace Name	Length (hours)	# of Disks	# of Requests	Average Size	Percent Reads	Percent Seq. Reads
Cello	168	8	3,262,824	6.3 KB	46.0%	2.5%
Snake	168	3	1,133,663	6.7 KB	52.4%	18.6%
Hplajw	168	2	44,519	5.9 KB	29.3%	0.2%
Air-Rsv	9	16	2,106,704	5.1 KB	79.3%	7.8%
Sci-TS	19.6	43	5,187,693	2.4 KB	81.5%	13.8%
Order	12	22	12,236,433	3.1 KB	86.2%	7.5%
Report	8	22	8,679,057	3.9 KB	88.6%	3.8%
TPCB1	0.1	9	32,768	2.3 KB	62.7%	0.005%
MultiWisc1	0.8	17	1,048,576	4.2 KB	95.8%	7.6%

Table A.1: Basic Characteristics of the Storage I/O Request Traces.

A.2.1.1 Current Library of Storage I/O Traces

Our research group has acquired a variety of storage I/O request traces for use in trace-driven storage subsystem simulation. Some of these traces have been generously shared with us by industry research partners; I gratefully acknowledge Hewlett-Packard Company and Digital Equipment Corporation. Others were collected from NCR systems, using the instrumentation described above. This would not have been possible without the support of NCR Corporation (now AT&T/GIS). While these traces are not used for the experiments reported in this dissertation, they are a crucial component of this simulation infrastructure and have been central to some of our previous storage subsystem research [Ganger93a, Worthington94]. Table A.1 lists basic statistics for these traces.

Three of the traces come from Hewlett-Packard systems running HP-UX, a version of the UNIX operating system. [Ruemmler93] describes these traces in detail. *Cello* comes from a server at HP Labs used primarily for program development, simulation, mail and news. *Snake* is from a file server used primarily for compilation and editing at the University of California, Berkeley. *Hplajw* comes from a personal workstation used mainly for electronic mail and editing papers. While each of these traces is actually two months in length, a single, week-long snapshot (5/30/92 to 6/6/92) is frequently used.

Four of the traces come from commercial VAX^{TM} systems running the VMS^{TM} operating system. [Rama92] describes these traces in detail. *Air-Rsv* is from a transaction processing environment in which approximately 500 travel agents made airline and hotel reservations. *Sci-TS* is from a scientific time-sharing environment in which analytic modeling software and graphical and statistical packages were used. *Order* and *Report* are from a machine parts distribution company. *Order* was collected during daytime hours, representing an order entry and processing workload. *Report* was collected at night, capturing a batch environment in which reports of the day's activities were generated.

Two sets of traces come from NCR systems executing database benchmarks with Oracle^{TM} database software. These traces were captured using the instrumentation described above and are described more thoroughly in [Ganger93a]. Several traces, named *TPCB#*, were captured on a workstation executing the TPC-B benchmark [TPCB90]. Other traces, named *MultiWisc#*, were captured on an 8-processor database server executing a multiuser database benchmark based on the Wisconsin benchmark [Gray91]. Several traces from each workload/configuration are kept because each trace is very short (6-50 minutes).

A.2.2 System-Level Traces

A system-level trace, as fed into the simulator, consists of three files:

1. **<tracename>.proc**: This file contains the process event sequences for all processes that were active at any point during the trace. Each event consists of a computation time, an event type and (optionally) additional event-specific fields.

2. **<tracename>.int**: This file contains the external interrupts, including the handler event sequences, in ascending arrival time order. Currently this file also contains the traced clock interrupts. Each entry contains the interrupt arrival time, the interrupt vector and the number of events in the interrupt handler. Interrupt handler events use the same format as process events.

3. **<tracename>.init**: This file contains two lists that are used to initialize simulator state. The first list identifies and defines all processes that were active during the trace. Each entry consists of the process ID number (PID), the index into the *<tracename>.proc* file and the number of events in the process's event sequence. The second list defines the initial state of all processes present in the system when the trace began (whether active or sleeping). Processes created during trace capture are not present in the second list. Each entry contains the PID, the parent process's PID and information related to process control, including the process's initial state (i.e., sleeping, running or waiting for the CPU).

Trace-Driven System-Level Simulation

As indicated above, the simulator emulates a system-level model when driven by system-level traces. During the initialization phase, the **<tracename>.init** file is used to establish the initial state of the process control module and the CPU module. The first event of each process that starts out executing on a CPU is read from the appropriate location in the *<tracename>.proc* file. CPUs that do not start out with an executing process begin in the idle loop. CPUs can not start out executing an interrupt handler. When a process event computation time reaches zero, the event "occurs" and the simulator performs two tasks: (1) updates the simulator state appropriately, and (2) reads the next event from the process trace file, unless the first step added new events to the process's event stream. When a context switch event "occurs," a new process "executes" on the CPU, reading events from the process trace file as needed. Also during the initialization phase, the first interrupt entry is read from the *<tracename>.int* file and added to the internal event queue. When an interrupt arrives, it takes over the CPU and "executes" its handler event sequence. At the same time, the next interrupt entry is read and added to the internal event queue.

System-Level Trace Acquisition

To collect traces of system activity, I instrumented the UNIX SVR4 MP operating system that executes on the experimental system described in the next section. The instrumentation collects trace data in a dedicated kernel memory buffer and alters performance by less than 0.1%, 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 data to a file. The trace record for each event contains the event type, the CPU number, a high-resolution timestamp and optional event-specific information. The timestamping mechanism (described in appendix C) combines a high-resolution (approximately 840 nanoseconds) diagnostic counter and the low-resolution (approximately 10 milliseconds) operating system "clock."[6]

A post-processing tool translates the traces of system activity into system-level traces that can be used with the simulator. To enable this, and also to provide configuration information, the traces capture 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 post-processing tool passes once through the trace and produces seven files:

1. <**tracename**>**.proc**: as described above.

2. <**tracename**>**.int**: as described above.

3. <**tracename**>**.init**: as described above.

4. <**tracename**>**.stats**: system usage/responsiveness statistics.

5. <**tracename**>**.valid**: ASCII version of traced system event sequence.

6. <**tracename**>**.ascii**: storage I/O request trace in the ASCII input format supported by the simulator.

7. <**tracename**>**.io**: trace of I/O initiations and interrupts, which is used for validation purposes only.

[6] The per-event timestamp information consists only of the diagnostic counter value. The first traced clock interrupt event captures the value of the operating system clock, which is incremented once for each clock interrupt. Since every clock interrupt is captured in the trace, the operating system clock value for any traced event can be computed. This approach increases the trace information that can be captured in a given quantity of buffer space.

A.2.2.1 Current Library of System-Level Traces

This information can be found in section 5.2.

A.2.3 Synthetic I/O Workload Descriptions

The simulator provides simple synthetic I/O request generators that can be used to drive storage subsystem simulations. The parameter file specifies the number of generators and the characteristics of each. The synthetic generators use the system-level model. Each generator is a process that executes within the system-level model, issuing storage I/O requests and, when appropriate, waiting for them to complete. The simulator's synthetic generators are currently configured with the following information:

- The number and effective capacity of the storage devices that can be accessed.

- The blocking factor — all request starting addresses and sizes must be a multiple of the blocking factor.

- Probabilities that a request is sequential or "local" to the last one issued by the generator process and a random variable definition[7] for the distance of a local request from the last starting address. The sum of the two probabilities must be less than or equal to one. Generating a starting address begins with determining whether it should be sequential, local or unrelated to the previous request. A sequential request starts at the address immediately following the last block previously accessed. A local request starts some distance, taken from the random variable definition mentioned above, from the previous starting address. An unrelated request randomly selects a device and starting address from the available storage space.

- A random variable definition for request sizes. For local and unrelated requests, the size is taken from this random variable definition. For sequential requests, the size is the same as the previous request.

- The probability that a request is a read (otherwise, it is a write).

- Random variable definitions for inter-request computation times. There are separate such definitions for sequential, local and unrelated requests.

- Probabilities that a request is time-critical or time-limited and a random variable definition for time limits. The sum of these probabilities must be less than or equal to one. Each request is time-critical, time-limited or time-noncritical (if neither of the others). The generated time limit represents the computation time before the generator process waits for the request to complete, which may differ from the resulting time limit due to CPU contention.

[7] In the simulator, a random variable definition consists of a distribution function (the type plus configuration values). The synthetic generator module currently supports five distribution functions: uniform, normal, exponential, poisson and two-step. The configuration values required depend upon the distribution function (e.g., the endpoints of a uniform distribution or the mean of an exponential distribution).

These generators can be configured to emulate the workload generators used in open and closed subsystem models, as well as a wide range of other options. An open subsystem model is achieved by setting the probabilities of time-critical and time-limited requests both to zero and configuring the system-level model to have the same number of processors as there are generators. The inter-request computation times will be thus be the inter-arrival times (per generator). A closed subsystem model is achieved by setting to one the probability that a request is time-critical. Setting the inter-request computation times to zero eliminates think times.

APPENDIX B

Disk Drive Module Implementation Details

Secondary storage performance is often dominated by disk drive behavior. This appendix describes the simulator's disk drive module, which accurately models mechanical positioning latencies, zoned recording, spare regions, defect slipping and reallocation, on-board buffers and caches, prefetch, fast write, bus delays, command queueing and request processing overheads. The first section of this appendix describes modern disk drives, providing both a short tutorial on the subject and insights into how the various components of a disk can be modeled. The second section describes the organization and operation of the disk drive module.

B.1 Modern Disk Drives

This section describes the main performance characteristics of modern disk drives. In many cases, approaches to modeling specific disk attributes are also discussed. The description focuses on common characteristics of disk drives conforming to the popular SCSI bus protocol [SCSI93]. Much of the discussion also applies to drives using other peripheral interfaces.

A disk drive can be viewed as two distinct parts, one physical and one logical. The physical part consists of the media recording and accessing components. The logical part consists of the on-board control logic that interprets requests, manages internal resources and improves the efficiency of media access.

B.1.1 Media Recording and Accessing

Mechanical Components

A disk drive (see figure B.1) consists of a set of **platters** rotating on a common axis. Both **surfaces** of each platter are coated with magnetic media. Each surface has an associated **read/write head**. In most drives, only one read/write head is active at any given time. The heads are mounted on disk **arms** that are ganged together on a common **actuator**.

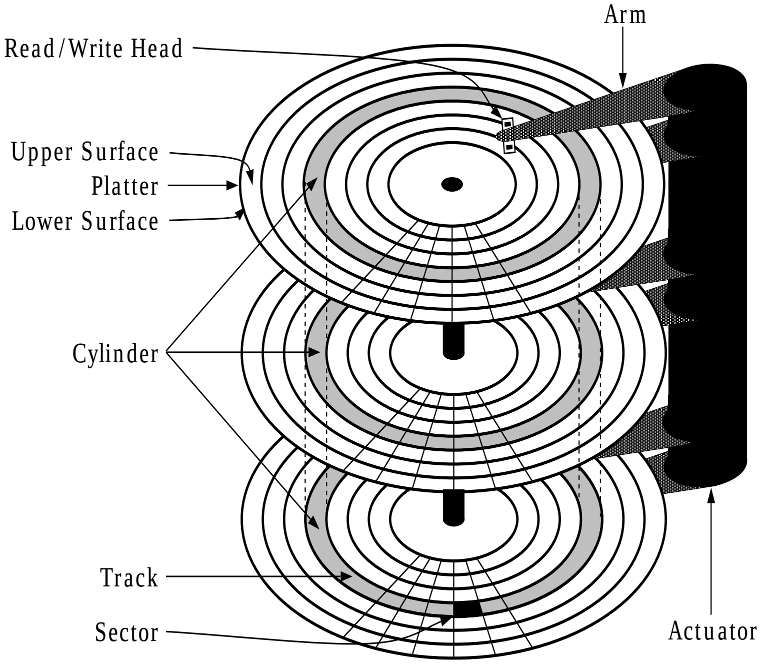


Figure B.1: Disk Drive Internals.

Data Layout

The minimum unit of data storage is a **sector**, which typically holds 512 bytes of data. Sectors are arranged in concentric circles, called **tracks**, on each surface.[1] A **cylinder** is a set of tracks (one from each surface) equidistant[2] from the center of the disk.

To exploit the larger circumference of outer tracks, a modern disk usually partitions its set of cylinders into multiple **zones**, or bands. The number of sectors per track increases with the radius of the tracks in the zone (see figure B.2). This increases both the capacity per unit volume and the mean and peak media transfer rates. However, it requires more powerful electronics to manage the varied recording/accessing activity. Also, additional on-board logic resources are needed to handle the more complex data layout.

Read/Write Head Positioning

Mechanical positioning delays often dominate disk request service times. To access a specific media location, a disk actuator must first **seek** to the target cylinder and activate the appropriate read/write head, possibly requiring a **head switch**. The read/write head then waits for the target sectors to rotate into position. After this **rotational latency**, the media transfer occurs as the target sectors rotate under the read/write head. Additional mechanical delays (e.g., head switches and/or seeks) result when multiple tracks or cylinders must be accessed.

[1] This data layout differs from the spiral layout used with compact disks and vinyl records.

[2] "Equidistant," in this case, refers to the per-surface track numbering. Thermal variations and other physical phenomenon may cause the Nth track on two surfaces to not be exactly the same distance from the center.

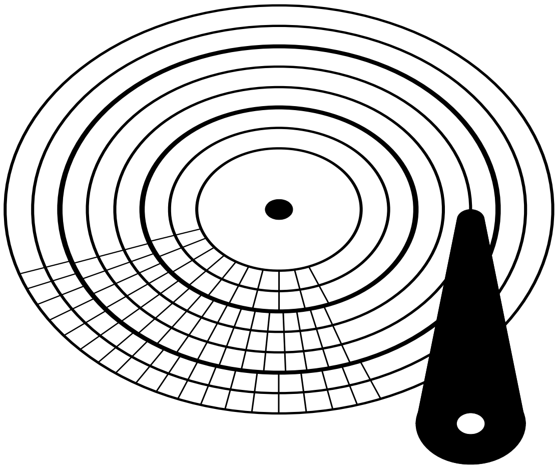


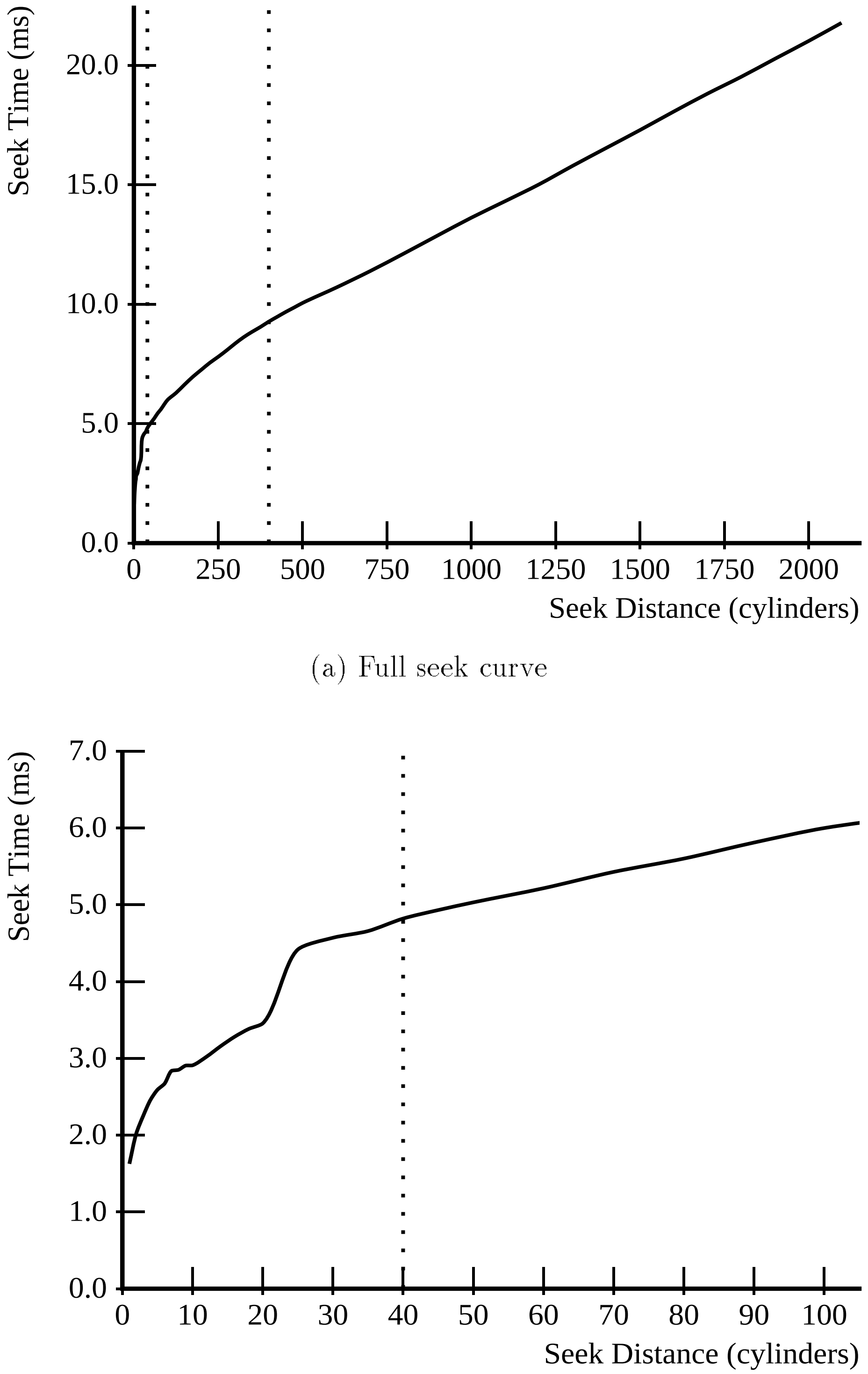
Figure B.2: Top View of a Disk Surface with 3 Zones. The innermost and outermost zone each contain 2 cylinders. The middle zone contains three.

A seek is comprised of four phases: (1) acceleration, (2) coast, (3) deceleration, and (4) settle. The disk arm accelerates until either the maximum velocity is reached or half of the seek distance has been traversed. If necessary, the disk arm coasts for some distance at its maximum velocity. The disk arm then decelerates, coming to rest near the destination track. Finally, the actuator electronics make fine adjustments to the disk arm position to **settle** precisely onto the track.[3] Very short seeks may be handled entirely by the settling electronics, without utilizing the full power of the actuator motor. Before writing data to the media, the electronics will often take additional **write settle time** to more accurately position to the center of the track. Data can be read with the read/write head slightly misaligned but writing data off-center invites cross-talk between tracks, complicating or even preventing later access.

A **seek curve** maps seek distances to actuator positioning times. In reality, the seek time between two tracks is a complex function of many variables, including the positions of the cylinders (i.e., their distances from the spindle), the direction of the seek (inward or outward), various environmental factors (e.g., vibration and thermal variation), the type of servo information (e.g., dedicated, embedded or hybrid), the mass and flexibility of the head/arm assembly and the current available to the actuator. However, a single curve can effectively reflect mean values for seek times. Typical seek curves (e.g., see figure B.3) consist of an irregular initial region (caused by variations in how the actuator electronics optimize specific seek distances), a middle region approximated by a square root function (caused by the acceleration/deceleration phases), and a large linear region for long seeks. A simulator can use a seek curve directly or in the form of some fitted function. The additional write settle time is added when appropriate.

A head switch consists of electronically disabling the input/output signals of the active read/write head, enabling those of another and settling onto the new track. The settling is required because the radii of the two corresponding tracks may not match exactly (e.g., due

[3] As the platters rotate, the same track-following electronics are used to constantly fine-tune the read/write head position.



(a) Full seek curve

(b) Expanded view of short seeks

Figure B.3: Measured seek curve for a Seagate ST41601N disk drive.

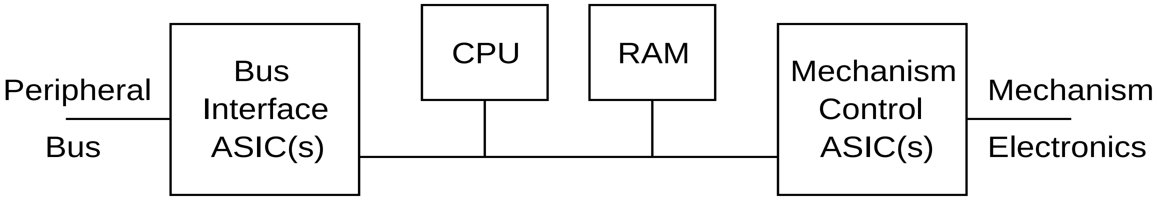


Figure B.4: On-Board Disk Drive Logic.

to thermal variation). The head switch time depends largely on the type of servo information utilized by the drive. Also, write settle time is required before writing data. When both a seek and a head switch are necessary, they usually occur concurrently. So, the combined delay is the maximum of the two.

The rotation speed, usually measured in rotations per minute (RPM), directly affects both the rotational latency and the media transfer rate. Generally, disk specifications indicate that the rotation speed varies slightly (e.g., ±0.5%) both among disks in a line of drives and over time for a given disk. Both forms of variation are generally small enough that they can be safely ignored in a disk model. However, the initial rotational offset of each drive in a simulation should be determined separately (e.g., by a uniform distribution), unless the the drives are intended to be rotationally synchronized (e.g., synchronous disk interleaving [Kim86]).

B.1.2 On-Board Control Logic

Figure B.4 shows the main on-board logic components used in modern disk drives. The CPU, which is often an embedded version of a previous generation commodity microprocessor (e.g., Intel 80386 or Motorola 68030), executes firmware to manage the disk's resources and handle requests. The firmware is often a simple operating system that provides process structures, interrupt service routines and simple memory and lock management. Like main memory in a computer system, the on-board memory is logically (and sometimes physically) partitioned into working space for the firmware and a staging area for the disk blocks moving to/from the media. Mechanism control ASIC(s) interact directly with the mechanical and magnetic disk components discussed above. Bus interface ASIC(s) interact directly with the peripheral bus, handling the details of arbitration, signal generation and interpretation, parity generation/checking, and so forth. The on-board logic components communicate via some interconnection, which is not necessarily a common bus.

The firmware executing on the CPU manages the other components. The various policies and functions of the firmware are discussed below.

External Interface Management

Most modern disk drives conform to a standardized, high-level peripheral bus[4] protocol (e.g., SCSI or IPI). A host or intermediate controller issues a request to a disk drive in terms of a starting **logical block number** (LBN) and a total request size. The details of the subsequent media access are hidden from the host or controller, offloading the management overhead associated with actual data storage.

The bus transfer rate and bus control delays are functions of the disk drive, the interconnect and the other participant in the communication (e.g., intermediate controller). A detailed disk performance model should account for the timings of control message exchange and data transfers to/from external components. This requires that the interconnect and other subsystem components be modeled appropriately.

From a performance standpoint, the four main bus interactions are request arrival, request completion, data transfer, and disconnect/reconnect. To issue a request to a disk drive, the controller establishes a connection to the disk and sends the appropriate message. When a request has been serviced, the disk establishes a connection and informs the controller. When the disk is ready, it initiates the data transfer to/from the controller.[5] Disconnect/reconnect actions can dramatically improve the effective utilization of a shared interconnect. With the SCSI protocol, components arbitrate for bus ownership and must explicitly give it up. Without disconnect/reconnect, the bus would be held for the duration of each request's service, preventing concurrent activity by other disks on the same bus. When a disk does not need the bus (e.g., during mechanical positioning) it can **disconnect** from the bus, allowing other components to use it. The disk later **reconnects** (i.e., re-arbitrates for bus ownership and re-establishes communication) when it needs to communicate with the controller.

In normal circumstances, a disk that has disconnected will reconnect for three reasons: (1) An outstanding write request has been serviced and the disk needs to report completion. (2) The disk is ready to transfer read request data to the controller. (3) The disk is ready to receive write request data from the controller. For SCSI disk drives, the decision as to when a disk is "ready" for bus data transfers (the latter two reasons above) is governed by two watermark values. The *Buffer Full Ratio* determines how much buffered (cached) read data should be available before reconnecting. The *Buffer Empty Ratio* determines how much dirty write data should remain before reconnecting (if additional write data need to be transferred). These ratios are usually specified as fractions of either the cache segment size (see below) or the request size. Poor settings can hurt performance by causing too many disconnect/reconnect cycles (there is overhead involved with each) or reducing the overlap between bus and media transfers. Some disks dynamically adjust the buffer full and empty ratios based on observed media and bus rates to avoid these problems.

[4] The peripheral interconnect may not necessarily be a shared bus. For example, some computer systems employ point-to-point serial connections. The physical aspects of the interconnect are not particularly important to performance modeling of disk drives, so long as the various delays and transfer rates are properly emulated by the bus module. The topology of the interconnect is an important but external issue (from the disk drive's point of view).

[5] Usually, the data are transferred in logically ascending order. However, some protocols allow blocks of data to be transferred in non-ascending order when it is more convenient for the disk.

Request Processing

The request processing firmware for a modern disk drive is large and complex. From a performance modeling standpoint, however, the two main aspects of this firmware are the policies and the processing times. The various policies are described under the other headings in this subsection. The request processing times can be an important part of the service time, even though they are generally dominated by mechanical positioning delays.

Initial request processing overheads (i.e., when the request arrives at the disk) can depend on several factors, including the request type (e.g., read or write), the request size, the state of the on-board cache (e.g., hit or miss) and the immediately previous request's characteristics[6]. Much of this initial overhead is visible externally because the controller passes bus ownership (implicitly) to the disk and the disk does not respond (e.g., by disconnecting or transferring data) until after some initial request processing. If the request is a read hit or a write, the data will usually be transferred to/from the controller immediately (i.e., without disconnecting). There are also externally visible processing delays associated with disconnect, reconnect and completion actions. Externally visible delays are particularly important when the bus is shared by more than one disk, since bus contention may occur. Request processing overheads that are not externally visible are also important, but can often be grouped with other service time components, such as seek times.

Logical-to-Physical Data Mappings

The traditional mapping of logical block numbers to physical media locations (i.e., cylinder, surface and sector) starts with logical block zero on the outermost (or innermost) cylinder. Logical block numbers are assigned around each track of the cylinder before moving to the next adjoining cylinder. This process repeats until the entire disk's capacity has been mapped. Several policies used in modern disk drive firmware disrupt the **LBN-to-PBN** mapping by withholding parts of the physical media for other purposes or shifting the mapping for performance purposes.

Many disk drives reserve a small fraction of the storage capacity for private use by the firmware. Generally, this reserved space is located at the beginning or end of a zone.

Because switching between surfaces and seeking between adjacent cylinders requires nonzero time, the first logical block of each track or cylinder is offset from the first logical block of the previous track or cylinder. This prevents a request that crosses a track or cylinder boundary from "just missing" the next logical block and waiting almost a full rotation for it to come around again. The **track skew** is the offset used when moving to the next track within a cylinder. The **cylinder skew** is the offset used when moving from the last track of one cylinder to the first track of the next. These values are specified as an integral number of sectors and therefore differ from zone to zone.

Disk drives can survive the loss of hundred of sectors or tracks to media corruption. In fact, a new drive usually contains a list of defects identified at the factory. Defects **grown** during a disk's lifetime are added to this list and removed from the LBN-to-PBN mapping. The firmware reserves **spare regions** at the end of certain tracks, cylinders or

[6] From empirical observation, it appears that disk drive firmware will sometimes "prime" itself to reduce command processing delays for common sequences of request types (e.g., sequential reads).

zones to replace defective media locations. The defect management firmware compensates for corrupt media location in one of two ways, slipping or relocation. Sector or track **slipping** occurs only during disk format. In the case of sector slipping, the format process skips each defective sector and adjusts the LBN-to-PBN mapping so that the sectors on either side are logically sequential. Track slipping follows the same routine, except that an entire track is skipped if it contains any defective media. With sector or track slipping, each defect changes the mapping of all subsequent logical blocks up to the next spare region. Sector or track **reallocation** occurs when a disk discovers defective media during normal use. The defect management firmware immediately remaps the corresponding LBN(s) to a spare region and directs subsequent accesses to the new location(s).

Because several layout and mapping characteristics vary from zone to zone, it is useful to partition the LBN-to-PBN mapping process into two parts. First, the correct zone is identified. Then, the physical location within the zone is identified. The first step can be realized with knowledge of the first logical block number of each zone. The second requires several zone-specific values, including the first cylinder number, the number of sectors per track, the track and cylinder skews, the amount and location of the reserved space and the physical sector that is assigned the first logical block number[7]. Given these values and information about spare regions and defect management, the mapping follows the traditional approach.

On-Board Cache

As mentioned above, part of the on-board memory serves as a staging area for disk blocks. In modern disk drives, this staging area acts both as a speed-matching buffer (between the bus and the media) and as a disk block cache. As a speed-matching buffer, the memory allows the firmware to overlap the transfers over the bus and to or from the media. As a disk block cache, the memory can improve effective disk service times. A request that can be satisfied by the cache avoids lengthy mechanical positioning delays. The service time for such a request can be more than an order of magnitude less than if media access were required. Modern disk drives use a wide variety of cache management policies, many of which are described below. Although the policies used by a given disk are rarely outlined in publically available specifications, there are approaches to identifying them empirically [Worthington95].

Most disks partition the available memory into multiple cache lines, referred to as **segments**, to better service multiple streams of sequential requests. Commonly, each segment is the same size. In some disks, this size matches the number of sectors per track. In most disks, however, there is no such relationship. The most aggressive implementations analyze request patterns and dynamically modify the number of cache segments (and thereby the segment size) to improve hit rates.

During service, each request is attached to a cache segment. When a new request arrives, the cache is scanned for any overlap. For read requests, an overlap indicates a partial or full

[7] In some disk drives, the physical sector that is assigned the first logical block varies from zone to zone in non-intuitive ways. Certainly, a **zone skew** would be useful for requests that cross zone boundaries, particularly when there is reserved space at the beginning or end of adjacent zones. Empirically, however, these values do not always correlate with the inter-zone seek times.

hit. For a full read hit, the data blocks are simply transferred from the cache to the controller. For a partial read hit, the firmware may utilize the data sectors or ignore them. For write requests, overlapping data blocks (possibly only the overlaps, possibly entire segments) are invalidated in the cache. The exception to this occurs with write-back caches (see below). For a cache miss, the firmware selects one of the segments, invalidate its contents and attaches it to the request. The segment replacement policy is often *Least-Recently-Used*. Some disks only allow write requests to use a subset of the segments and may prevent read requests from using that subset.

To improve the cache hit rate for sequential access patterns, most modern disk drives prefetch data after read requests. Specifically, the firmware continues reading sequentially numbered media blocks beyond the end of a read request. There are several possible stop conditions. Most disks discontinue prefetch to service a new request arrival or another pending request (for command queued disks; see below). Although the prefetch may be stopped when it reaches the end of a track or cylinder, most disks aggressively prefetch beyond these boundaries even though a new request might have to wait for the head switch or seek to complete. Some disks are configured with a maximum prefetch size. Finally, prefetch stops when the cache segment is full.

Disks with **read-on-arrival** and/or **write-on-arrival** capability can begin transferring data from or to the media as soon as any seek or head switch activity completes. That is, media sectors are transferred from or to the media in the order that the read/write head encounters them rather than in strictly ascending LBN order. This essentially eliminates rotational latency for full track requests,[8] but is less effective for smaller or larger requests. Read-on-arrival can also increase the effectiveness of prefetch for heavy workloads. Most modern disk drives do not currently support these options.

Some disks support the ability to signal write request completion as soon as all data blocks reach the on-board cache. The firmware moves the data from the cache to the media at some later point. Because the on-board memory of most disk drives is volatile (i.e., the contents are lost whenever power fails), such **fast writes** reduce the reliability of disk drive storage.[9]. Modern disks can employ three options to reduce the probability of data corruption due to improper update ordering.[10] One option is to disable fast write entirely. A second option, used in HP-UX [Ruemmler93], is to selectively disable fast write for each request that may require ordering with respect to a later request. This requires that the software executing on host systems recognizes that a disk uses fast write and tags each request appropriately. A third option is to always perform fast writes in arrival order. Many disks do not use fast write for a second consecutive write request until the first's data have been written to the disk, unless the new request is immediately sequential. If it is sequential, then it is combined with the previous request and reported as complete.

[8] For this reason, read-on-arrival and write-on-arrival are sometimes referred to as **zero-latency** read and write.

[9] In the near future, disk drives are likely to be equipped with non-volatile cache memory to eliminate this problem.

[10] Some applications and system software components rely on careful ordering of permanent storage updates to protect the data integrity. For example, databases add entries to a write-ahead log to achieve an atomic update to multiple records. Also, many file systems require ordered updates to maintain metadata integrity [Ganger94].

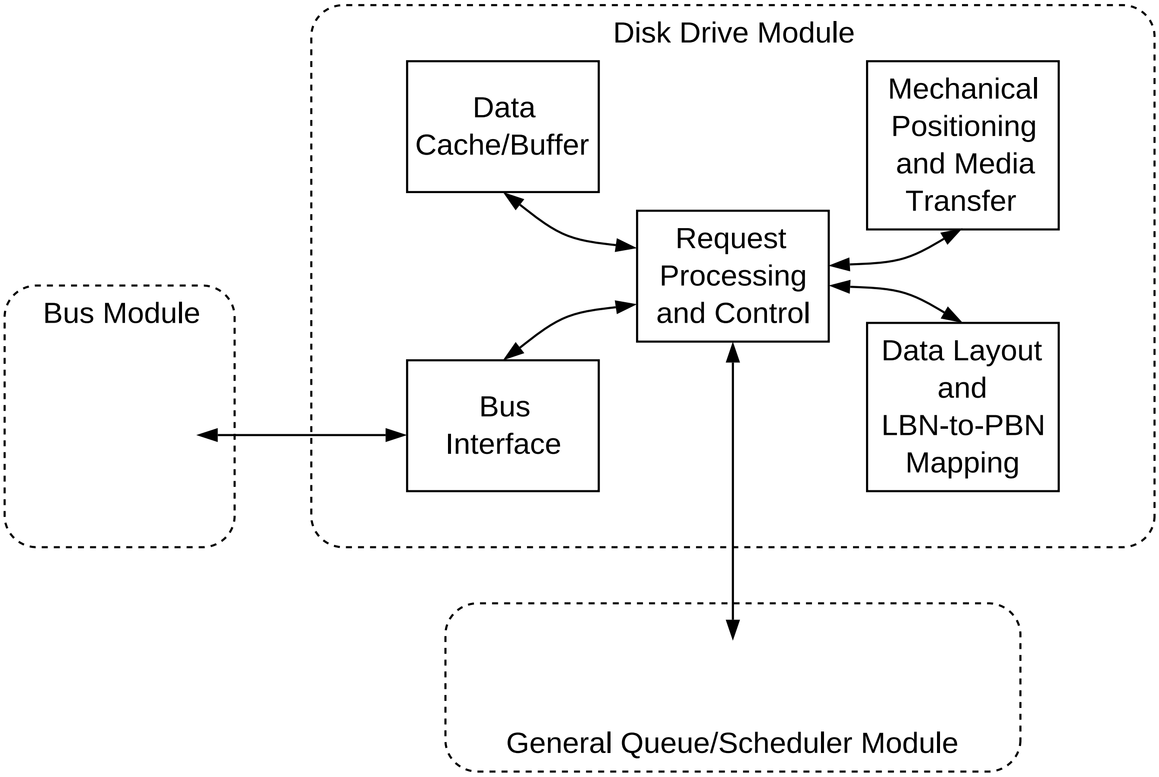

Disk Drive Module
Data Cache/Buffer
Mechanical Positioning and Media Transfer
Request Processing and Control
Bus Module
Data Layout and LBN-to-PBN Mapping
Bus Interface
General Queue/Scheduler Module

Figure B.5: Disk Drive Module.

Command Queueing

Most modern disk drives support **command queueing**, whereby some number (e.g., 2-64) of requests can be outstanding at a time. Command queueing has several benefits. First, an on-board scheduler can have an advantage over external schedulers because it generally has more accurate information about the current read/write head position and various positioning delays. The SCSI protocol provides for optional ordering restrictions, such as *Head-Of-Queue* and *Ordered* requests [SCSI93]. Second, the firmware can exploit inter-request concurrency by decoupling the command processing, bus transfer, and media access components of request service. Of course, there are dependencies between these components (e.g., write data cannot be transferred to the media before it is received from the host or controller), but the disk can still exploit concurrency between separate requests [Worthington95a].

B.2 Disk Drive Module

The disk drive module models all of the aspects of modern disk drives described in section B.1. It is organized as five submodules (see figure B.5) and uses instances of the general queue/scheduler module (see section A.1.2). This section describes the five submodules and their interactions.

Request Processing and Control

This submodule controls the flow of work in the disk module. It uses the other four submodules to accomplish this task. All of the disk service time components are decided here and performed using the general infrastructure described in section A.1.1. Any request concurrency (e.g., overlapping bus and media transfers) is also determined and handled by this submodule. Instances of the general queue/scheduler module are used to maintain the list of outstanding requests and make scheduling decisions. Most of the policies, for both this submodule and the instances of the queue/scheduler module, are configured by input parameters.

Data Cache/Buffer

This submodule maintains the state of the on-board cache segment(s). The control submodule uses it to determine whether requests hit (partially or entirely) or miss in the cache and to select a segment for staging a request's data. The control submodule makes all decisions regarding how to deal with cache hits and transfers into and out of a segment, simply informing the cache/buffer submodule when the contents change. The segment selection and replacement policies are configured by initialization parameters.

Mechanical Positioning and Media Transfer

This submodule maintains state regarding the position of the active read/write head. Also, the control submodule uses this submodule to determine how long any given mechanical positioning delay or media transfer will take. When appropriate, the control submodule informs this submodule that the current position should be updated. The current position state consists of the current cylinder, the current surface, the last known rotational offset and the time at which the offset held. This state is sufficient to determine the exact read/write head position at any point in simulated time (assuming that the rotation speed is constant). The rotation speed and various mechanical delay components are all input parameters.

Data Layout and LBN-to-PBN Mapping

This submodule simply translates logical block numbers to physical media locations (or the reverse). All of the various data layout and mapping characteristics are input parameters.

Bus Interface

This submodule interacts with the bus module. The control submodule uses this submodule to communicate with external components. The control submodule determines when ownership should be established/relinquished and when/which messages and data should be transferred.

APPENDIX C

A High-Resolution Timestamp Mechanism

Effective measurement of computer system activity, such as tracing disk requests and/or other system software events, requires long-duration, high-resolution timestamps. Duration refers to the maximum difference between two timestamps that can be effectively measured without external assistance. Resolution refers to the smallest granularity difference between any two timestamps. This appendix describes a mechanism for constructing sub-microsecond resolution timestamps on NCR System 3000 Level 3, 4 and 5 machines. This timestamping mechanism is an integral component of the system instrumentation described in sections A.2.1 and A.2.2. It has been implemented and tested on both a desktop NCR 3433 workstation and an 8-processor NCR 3550 system.

Each timestamp consists of two values, which are sampled copies of the internal system software clock and a small diagnostic counter. The former provides long-duration (over a year) time measurements at a granularity of approximately 10 milliseconds. The latter provides short-duration (about 55 milliseconds) time measurements at a granularity of approximately 850 nanoseconds. The two values are combined to achieve both long-duration and high-resolution.

This appendix is organized as follows. The next section describes the two timestamp components in more detail, including how they are initialized and updated. Section C.2 describes how the two timestamp components are combined to determine the relative time between any pair of timestamps.

C.1 Timestamp Components

C.1.1 System Software Clock

The system software maintains a rough view of time, relative to system initialization, via a single 32-bit unsigned integer referred to as the system software clock. This integer is set to zero during system initialization and is incremented once for each clock interrupt (i.e., during the clock interrupt service routine). Clock interrupts are generated by logic attached to the local peripheral bus (LPB) [NCR91]. The relevant logic contains two 16-bit counters (a working counter and a restart value), a high-frequency (1.1925 MHz) oscillating clock and some control bits. The control bits are set so that the working counter decrements once per clock cycle. When the working counter's value reaches zero, a clock interrupt is generated, the working counter is reset to the restart value and the process repeats. The default restart value for these NCR systems is 11932. With this value, a clock interrupt is generated approximately once every 10 ms. So, the system software clock's duration exceeds a year with a resolution of 10 ms.

While it would be convenient to realize timestamps by coupling the system software clock and the counter that generates the clock interrupts, it is not feasible in these systems. Accurate timestamps would require that resetting the counter and incrementing the system software clock be an effectively atomic action. That is, the two must occur atomically or the system software must know (when the timestamp is taken) whether or not the latter has occurred for the most recent instance of the former. The asynchronous operation of the counter makes this difficult. The architectures of the systems in question make it impossible. It is not possible to determine, from within the system software, whether or not a clock interrupt is pending. Further, in the multi-processor systems, it is not possible to determine whether or not an executing clock interrupt service routine has incremented the system software clock yet. These two factors would open the door for large (10 millisecond) timing errors.

C.1.2 Diagnostic Counter

The LPB logic discussed above includes three counters in addition to the one that generates clock interrupts. Three of the counters are clocked by the same high-frequency oscillator. The fourth, called the watchdog timer, uses the previously described counter's output as a clock signal that is gated by the clock interrupt pending bit. So, this counter is decremented only when a second clock interrupt is generated before the previous one has been serviced. When the working value reaches zero, a non-maskable interrupt is generated, forcing the system software to recognize that interrupts have been masked for too long. One of the additional counters controls the speaker tone generator. The other is used by the system diagnostic routines, but is not otherwise utilized during normal system operation.

This diagnostic counter can be safely used without interfering with system functions. Its behavior and construction are similar to those described above for the clock interrupt generator. The relevant logic contains a 16-bit working counter, a restart value and control bits. For timing purposes, we modified the system software to initialize the control bits and the restart value. Thereafter, the working counter repeatedly cycles through its full range, from 65535 to 0, without generating any interrupts. When the value reaches zero, the restart value (65535) is immediately reloaded. The duration of the counter is therefore 65535 cycles of the clock, or approximately 55 milliseconds, and the resolution is the inverse of the clock frequency (i.e., 838.574 nanoseconds).

C.2 Timestamp Conversion

A single timestamp consists of sampled values of the system software clock and the diagnostic counter. Both of these system components are independent of wall-clock time in that they are reset to static values during system initialization. Therefore, a single timestamp provides no useful information. However, the elapsed time between any two timestamps can be determined by manipulating the corresponding values, combining the best quality of each component (i.e., duration and resolution). In particular, the diagnostic counter values are used to identify the fine-grain difference between the timestamps. The system software clock values are used to determine how many times the diagnostic counter wrapped between the two timestamps. The following pseudo-code demonstrates the process:

```
int finediff =       timestamp1.hires $-$ timestamp2.hires
int loresdiff =      timestamp2.lores $-$ timestamp1.lores
int turnovers =      round(((loresdiff $\times$ 11932) $-$ finediff) $\div$ 65535)
int hiresticks =     finediff $+$ (turnovers $\times$ 65535)
float millisecs =    hiresticks $\times$ 0.838574
```

This combination of a long-duration, low-resolution values and short-duration, high-resolution values succeeds without error so long as the duration of the latter more than doubles the maximum error in the former. The resolution of the system software clock places a lower bound on the corresponding error (i.e., ±10 milliseconds). In addition, a clock interrupt can wait for service after it is generated. This occurs when the system software masks interrupts while executing critical sections of code. However, such critical sections are kept small to prevent excessive delays in handling interrupts. Also, the watchdog timer described above prevents lengthy delays in handling clock interrupts. Theoretically, a clock interrupt could be pending until the generation of the next. In practice, they are serviced soon after generation. However, the theoretical maximum delay means that differences in the system software clock could underpredict elapsed times by slightly more than 20 milliseconds. The diagnostic counter's duration of 55 milliseconds more than doubles this maximum error.[1]

[1] The timestamping mechanism requires that every clock interrupt be serviced by the system software. That is, no clock interrupts may be dropped. The small size of critical sections and the watchdog timer guarantee this condition.

BIBLIOGRAPHY

[Abbott90] R. Abbott, H. Garcia-Molina, "Scheduling I/O Request with Deadlines: a Performance Evaluation", *IEEE Real-Time Systems Symposium*, 1990, pp. 113–124.

[Amdahl67] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", *AFIPS Spring Joint Computing Conference*, Vol. 30, April 1967, pp. 483–485.

[Baker91] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, J. Ousterhout, "Measurements of a Distributed File System", *ACM Symposium on Operating Systems Principles*, 1991, pp. 198–212.

[Bennett94] S. Bennett, D. Melski, "A Class-Based Disk Scheduling Algorithms: Implementation and Performance Study", Unpublished Report, University of Wisconsin, Madison, 1994.

[Biswas93] P. Biswas, K.K. Ramakrishnan, D. Towsley, "Trace Driven Analysis of Write Caching Policies for Disks", *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1993, pp. 13–23.

[Bitton88] D. Bitton, J. Gray, "Disk Shadowing", *International Conference on Very Large Data Bases*, September 1988, pp. 331–338.

[Bitton89] D. Bitton, "Arm Scheduling in Shadowed Disks", *IEEE COMPCON*, Spring 1989, pp. 132–136.

[Brandwajn94] A. Brandwajn, D. Levy, "A Study of Cached RAID-5 I/O", *Computer Measurement Group (CMG) Conference*, 1994, pp. 393–403.

[Busch85] J. Busch, A. Kondoff, "Disc Caching in the System Processing Units of the HP 3000 Family of Computers", *HP Journal*, Vol. 36, No. 2, February 1985, pp. 21–39.

[Cao93] P. Cao, S. Lim, S. Venkataraman, J. Wilkes, "The TickerTAIP Parallel RAID Architecture", *IEEE International Symposium on Computer Architecture*, May 1993, pp. 52–63.

[Cao95] P. Cao, E. Felton, A. Karlin, K. Li, "A Study of Integrated Prefetching and Caching Strategies", *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1995, pp. 188-197.

[Carey89] M. Carey, R. Jauhari, M. Livny, “Priority in DBMS Resource Scheduling”, *International Conference on Very Large Data Bases*, 1989, pp. 397–410.

[Carson92] S. Carson, S. Setia, “Analysis of the Periodic Update Write Policy for Disk Cache”, *IEEE Transactions on Software Engineering*, Vol. 18, No. 1, January 1992, pp. 44–54.

[Chen90] P. Chen, D. Patterson, “Maximizing throughput in a striped disk array”, *IEEE International Symposium on Computer Architecture*, 1990, pp. 322–331.

[Chen90a] P. Chen, G. Gibson, R. Katz, D. Patterson, “An Evaluation of Redundant Arrays of Disks using an Amdahl 5890”, *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1990, pp. 74–85.

[Chen91] S. Chen, D. Towsley, “A Queueing Analysis of RAID Architectures”, University of Massachusetts, Amherst, COINS Technical Report 91–71, September 1991.

[Chen91a] S. Chen, J. Kurose, J. Stankovic, D. Towsley, “Performance Evaluation of Two New Disk Request Scheduling Algorithms for Real-Time Systems”, *Journal of Real-Time Systems*, Vol. 3, 1991, pp. 307–336.

[Chen93a] P. Chen, E. Lee, A. Drapeau, K. Lutz, E. Miller, S. Seshan, K. Sherriff, D. Patterson, R. Katz, “Performance and Design Evaluation of the RAID-II Storage Server”, *International Parallel Processing Symposium, Workshop on I/O*, 1993.

[Chen93b] P. Chen, E. Lee, G. Gibson, R. Katz, D. Patterson, “RAID: High-Performance, Reliable Secondary Storage”, *ACM Computing Surveys*, Vol. 26, No. 2, June 1994, pp. 145–188.

[Chen95] P. Chen, E. Lee, “Striping in a RAID Level 5 Disk Array”, *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1995, pp. 136–145.

[Chervenak91] A.L. Chervenak, R. Katz, “Performance of a RAID Prototype”, *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1991, pp. 188–197.

[Chiu78] W. Chiu, W. Chow, “A Performance Model of MVS”, *IBM System Journal*, Vol. 17, No. 4, 1978, pp. 444–463.

[Coff72] E. G. Coffman, L. A. Klimko, B. Ryan, “Analysis of Scanning Policies for Reducing Disk Seek Times”, *SIAM Journal of Computing*, Vol. 1, No. 3, September 1972, pp. 269–279.

[Coffman82] E. Coffman, M. Hofri, “On the Expected Performance of Scanning Disks”, *SIAM Journal of Computing*, Vol. 11, No. 1, February 1982, pp. 60–70.

[Copeland89] G. Copeland, T. Keller, “A Comparison of High-Availability Media Recovery Techniques”, *ACM SIGMOD International Conference on Management of Data*, 1989, pp. 98–109.

[Dan94] A. Dan, D. Dias, P. Yu, "Buffer Analysis for a Data Sharing Environment with Skewed Data Access", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 6, No. 2, April 1994, pp. 331–337.

[Daniel83] S. Daniel, R. Geist, "V-SCAN: An adaptive disk scheduling algorithm", *IEEE International Workshop on Computer Systems Organization*, March 1983, pp. 96–103.

[Denning67] P. J. Denning, "Effects of scheduling on file memory operations", *AFIPS Spring Joint Computer Conference*, April 1967, pp. 9–21.

[Ebling94] M. Ebling, M. Satyanarayanan, "SynRGen: An Extensible File Reference Generator", *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1994, pp. 108–117.

[English91] R. English, A. Stepanov, "Loge: A Self-Organizing Disk Controller", Hewlett-Packard Laboratories Report, HPL-91-179, December 1991.

[Ferr84] D. Ferrari, "On the Foundation of Artificial Workload Design", *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1984, pp. 8-14.

[Ganger93] G. Ganger, Y. Patt, "The Process-Flow Model: Examining I/O Performance from the System's Point of View", *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1993, pp. 86–97.

[Ganger93a] G. Ganger, B. Worthington, R. Hou, Y. Patt, "Disk Subsystem Load Balancing: Disk Striping vs. Conventional Data Placement", *Hawaii International Conference on System Sciences*, January 1993, pp. 40–49.

[Ganger94] G. Ganger, Y. Patt, "Metadata Update Performance in File Systems", *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994, pp. 49–60.

[Ganger94a] G. Ganger, B. Worthington, R. Hou, Y. Patt, "Disk Arrays: High Performance, High Reliability Storage Subsystems", *IEEE Computer*, Vol. 27, No. 3, March 1994, pp. 30–36.

[Geist87] R. Geist, S. Daniel, "A Continuum of Disk Scheduling Algorithms", *ACM Transactions on Computer Systems*, Vol. 5, No. 1, February 1987, pp. 77–92.

[Geist87a] R. Geist, R. Reynolds, E. Pittard, "Disk Scheduling in System V", *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1987, pp. 59–68.

[Geist94] R. Geist, J. Westall, "Disk Scheduling in Linux", *Computer Measurement Group (CMG) Conference*, December 1994, pp. 739–746.

[Gingell87] R. Gingell, J. Moran, W. Shannon, "Virtual Memory Architecture in SunOS", *Summer USENIX Conference*, June 1987, pp. 81–94.

[Golding95] R. Golding, P. Bosch, C. Staelin, T. Sullivan, J. Wilkes, "Idleness is not sloth", *Winter USENIX Conference*, January 1995, pp. 201–22.

[Gotl73] C. C. Gotlieb, G. H. MacEwen, "Performance of Movable-Head Disk Storage Devices", *Journal of the Association for Computing Machinery*, Vol. 20, No. 4, October 1973, pp. 604–623.

[Gray91] ed. J. Gray, *The Benchmark Handbook for Database and Transaction Processing Systems*, Morgan Kaufman Publishers, San Mateo, CA, 1991.

[Griffioen94] J. Griffioen, R. Appleton, "Reducing File System Latency using a Predictive Approach", *Summer USENIX Conference*, June 1994, pp. 197-207.

[Hagmann87] R. Hagmann, "Reimplementing the Cedar File System Using Logging and Group Commit", *ACM Symposium on Operating Systems Principles*, November 1987, pp. 155–162.

[Haigh90] P. Haigh, "An Event Tracing Method for UNIX Performance Measurement", *Computer Measurement Group (CMG) Conference*, 1990, pp. 603–609.

[Holland92] M. Holland, G. Gibson, "Parity Declustering for Continuous Operation in Redundant Disk Arrays", *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992, pp. 23–35.

[Hofri80] M. Hofri, "Disk Scheduling: FCFS vs. SSTF Revisited", *Communications of the ACM*, Vol. 23, No. 11, November 1980, pp. 645–653.

[Hospodor94] A. Hospodor, "The Effect of Prefetch in Caching Disk Buffers", Ph.D. Dissertation, Santa Clara University, 1994.

[Hou93] R. Hou, Y. Patt, "Comparing Rebuild Algorithms for Mirrored and RAID5 Disk Arrays", *ACM SIGMOD International Conference on Management of Data*, May 1993, pp. 317–326.

[Hou93a] R. Hou, J. Menon, Y. Patt, "Balancing I/O Response Time and Disk Rebuild Time in a RAID5 Disk Array", *Hawaii International Conference on System Sciences*, January 1993, pp. 70–79.

[Hou93b] R. Hou, Y. Patt, "Trading Disk Capacity for Performance", *International Symposium on High-Performance Distributed Computing*, July 1993, pp. 263–270.

[Houtekamer85] G. Houtekamer, "The Local Disk Controller", *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1985, pp. 173–182.

[HP92] Hewlett-Packard Company, "HP C2244/45/46/47 3.5-inch SCSI-2 Disk Drive Technical Reference Manual", Part Number 5960-8346, Edition 3, September 1992.

[HP93] Hewlett-Packard Company, "HP C2490A 3.5-inch SCSI-2 Disk Drives, Technical Reference Manual", Part Number 5961-4359, Edition 3, September 1993.

[HP94] Hewlett-Packard Company, "HP C3323A 3.5-inch SCSI-2 Disk Drives, Technical Reference Manual", Part Number 5962-6452, Edition 2, April 1994.

[Hsiao90] H. Hsiao, D. DeWitt, "Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines", *IEEE International Conference on Data Engineering*, 1990, pp. 456–465.

[Jacobson91] D. Jacobson, J. Wilkes, "Disk Scheduling Algorithms Based on Rotational Position", Hewlett-Packard Technical Report, HPL-CSP-91-7, February 26, 1991.

[Karedla94] R. Karedla, J. S. Love, B. Wherry, "Caching Strategies to Improve Disk System Performance", *IEEE Computer*, Vol. 27, No. 3, March 1994, pp. 38–46.

[Kim86] M. Kim, "Synchronized Disk Interleaving", *IEEE Transactions on Computers*, Vol. C-35, No. 11, November 1986, pp. 978–988.

[Kim91] M. Kim, "Asynchronous Disk Interleaving: Approximating Access Delays", *IEEE Transactions on Computers*, Vol. 40, No. 7, July 1991, pp. 801–810.

[Kotz94] D. Kotz, S. Toh, S. Radhakrishnan, "A Detailed Simulation Model of the HP 97560 Disk Drive", Report No. PCS-TR94-220, Dartmouth College, July 18, 1994.

[Lary93] R. Lary, Storage Architect, Digital Equipment Corporation, Personal Communication, February 1993.

[Lee91] E. Lee, R. Katz, "Peformance Consequences of Parity Placement in Disk Arrays", *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 190–199.

[Lee93] E. Lee, "Performance Modeling and Analysis of Disk Arrays", Ph.D. Dissertation, University of California, Berkeley, 1993.

[Livny87] M. Livny, S. Khoshafian, H. Boral, "Multi-Disk Management Algorithms", *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1987, pp. 69–77.

[McKusick84] M. McKusick, W. Joy, S. Leffler, R. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 181–197.

[McVoy91] L. McVoy, S. Kleiman, "Extent-like Performance from a UNIX File System", *Winter USENIX Conference*, January 1991, pp. 1–11.

[Menon91] J. Menon, D. Mattson, "Performance of Disk Arrays in Transaction Processing Environments", IBM Research Report RJ 8230, July 15, 1991.

[Menon92] J. Menon, J. Kasson, "Methods for Improved Update Performance of Disk Arrays", *Hawaii International Conference on System Sciences*, January 1992, pp. 74–83.

[Menon93] J. Menon, J. Cortney, “The Architecture of a Fault-Tolerant Cached RAID Controller”, *IEEE International Symposium on Computer Architecture*, May 1993, pp. 76–86.

[Miller91] E. Miller, R. Katz, “Input/Output Behavior of Supercomputing Applications”, *Supercomputing*, 1991, pp. 567–576.

[Mitsuishi85] A. Mitsuishi, T. Mizoguchi, T. Miyachi, “Performance Evaluation for Buffer-Contained Disk Units”, *Systems and Computers in Japan*, Vol. 16, No. 5, 1985, pp. 32–40.

[Miyachi86] T. Miyachi, A. Mitsuishi, T. Mizoguchi, “Performance Evaluation for Memory Subsystem of Hierarchical Disk-Cache”, *Systems and Computers in Japan*, Vol. 17, No. 7, 1986, pp. 86–94.

[Mogul94] J. Mogul, “A Better Update Policy”, *Summer USENIX Conference*, 1994, pp. 99–111.

[Moran87] J. Moran, “SunOS Virtual Memory Implementation”, *EUUG Conference*, Spring 1988, pp. 285–300.

[Mourad93] A. Mourad, W.K. Fuchs, D. Saab, “Performance of Redundant Disk Array Organizations in Transaction Processing Environments”, *International Conference on Parallel Processing*, Vol. I, 1993, pp. 138–145.

[Muchmore89] S. Muchmore, “A comparison of the EISA and MCA architectures”, *Electronic Engineering*, March 1989, pp. 91–97.

[Mummert95] L. Mummert, M. Ebling, M. Satyanarayanan, “Exploiting Weak Connectivity for Mobile File Access”, Unpublished Report, Carnegie Mellon University, March 1995.

[Muntz90] R. Muntz, J. Lui, “Performance Analysis of Disk Arrays Under Failure”, *International Conference on Very Large Data Bases*, 1990, pp. 162–173.

[Myers86] G. Myers, A. Yu, D. House, “Microprocessor Technology Trends”, *Proceedings of the IEEE*, Vol. 74, December 1986, pp. 1605–1622.

[NCR90] NCR Corporation, “Using the 53C700 SCSI I/O Processor”, SCSI Engineering Notes, No. 822, Rev. 2.5, Part No. 609-3400634, February 1990.

[NCR91] NCR Corporation, “Class 3433 and 3434 Technical Reference”, Document No. D2-0344-A, May 1991.

[Ng88] S. Ng, D. Lang, R. Selinger, “Trade-offs Between Devices and Paths In Achieving Disk Interleaving”, *IEEE International Symposium on Computer Architecture*, 1988, pp. 196–201.

[Ng92] S. Ng, R. Mattson, “Maintaining Good Performance in Disk Arrays During Failure Via Uniform Parity Group Distribution”, *International Symposium on High-Performance Distributed Computing*, September 1992, pp. 260–269.

[Ng92a] S. Ng, "Prefetch Policies For Striped Disk Arrays", IBM Research Report RJ 9055, October 23, 1992.

[Oney75] W. Oney, "Queueing Analysis of the Scan Policy for Moving-Head Disks", *Journal of the ACM*, Vol. 22, No. 3, July 1975, pp. 397–412.

[Orji93] C. Orji, J. Solworth, "Doubly Distorted Mirrors", *ACM SIGMOD International Conference on Management of Data*, May 1993, pp. 307–316.

[Ouchi78] N. Ouchi, "System for Recovering Data Stored in Failed Memory Unit", U.S. Patent #4,092,732, May 30, 1978.

[Ousterhout85] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, J. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System", *ACM Symposium on Operating System Principles*, 1985, pp. 15–24.

[Ousterhout90] J. Ousterhout, "Why Aren't Operating Systems Getting Faster As Fast as Hardware?", *Summer USENIX Conference*, June 1990, pp. 247–256.

[Patterson88] D. Patterson, G. Gibson, R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *ACM SIGMOD International Conference on Management of Data*, May 1988, pp. 109–116.

[Patterson93] R.H. Patterson, G. Gibson, M. Satyanarayanan, "A Status Report on Research in Transparent Informed Prefetching", *ACM Operating Systems Review*, Vol. 27, No. 2, April 1993, pp. 21–34.

[Rama92] K. Ramakrishnan, P. Biswas, R. Karelda, "Analysis of File I/O Traces in Commercial Computing Environments", *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1992, pp. 78–90.

[Reddy89] A.L.N. Reddy, P. Banerjee, "An Evaluation of Multiple-Disk I/O Systems", *IEEE Transactions on Computers*, Vol. 38, No. 12, December 1989, pp. 1680–1690.

[Reddy92] A.L.N. Reddy, "A Study of I/O System Organizations", *IEEE International Symposium on Computer Architecture*, May 1992, pp. 308–317.

[Richardson92] K. Richardson, M. Flynn, "TIME: Tools for Input/Output and Memory Evaluation", *Hawaii International Conference on Systems Sciences*, January 1992, pp. 58–66.

[Ritchie86] D. Ritchie, "The UNIX I/O System", UNIX User's Supplementory Document, University of California, Berkeley, April 1986.

[Ruemmler91] C. Ruemmler, J. Wilkes, "Disk Shuffling", Technical Report HPL-CSP-91-30, Hewlett-Packard Laboratories, October 3, 1991.

[Ruemmler93] C. Ruemmler, J. Wilkes, "UNIX Disk Access Patterns", *Winter USENIX Conference*, January 1993, pp. 405–420.

[Ruemmler93a] C. Ruemmler, J. Wilkes, "A Trace-Driven Analysis of Disk Working Set Sizes", Technical Report HPL-OSR-93-23, Hewlett-Packard Laboratories, April 1993.

[Ruemmler94] C. Ruemmler, J. Wilkes, "An Introduction to Disk Drive Modeling", *IEEE Computer*, Vol. 27, No. 3, March 1994, pp. 17–28.

[Salem86] K. Salem, G. Garcia-Molina, "Disk Striping", *IEEE International Conference on Data Engineering*, 1986, pp. 336–342.

[Sandberg85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, "Design and Implementation of the Sun Network Filesystem", *Summer USENIX Conference*, June 1985, pp. 119–130.

[Satya86] M. Satyanarayanan, *Modeling Storage Systems*, UMI Research Press, Ann Arbor, MI, 1986.

[SCSI93] "Small Computer System Interface-2", ANSI X3T9.2, Draft Revision 10k, March 17, 1993.

[Seagate92] Seagate Technology, Inc., "SCSI Interface Specification, Small Computer System Interface (SCSI), Elite Product Family", Document Number 64721702, Revision D, March 1992.

[Seagate92a] Seagate Technology, Inc., "Seagate Product Specification, ST41600N and ST41601N Elite Disc Drive, SCSI Interface", Document Number 64403103, Revision G, October 1992.

[Seaman66] P. H. Seaman, R. A. Lind, T. L. Wilson "An analysis of auxiliary-storage activity", *IBM System Journal*, Vol. 5, No. 3, 1966, pp. 158–170.

[Seaman69] P. Seaman, R. Soucy, "Simulating Operating Systems", *IBM System Journal*, Vol. 8, No. 4, 1969, pp. 264–279.

[Seltzer90] M. Seltzer, P. Chen, J. Ousterhout, "Disk Scheduling Revisited", *Winter USENIX Conference*, 1990, pp. 313–324.

[Seltzer93] M. Seltzer, K. Bostic, M. McKusick, C. Staelin, "An Implementation of a Log-Structured File System for UNIX", *Winter USENIX Conference*, January 1993, pp. 201–220.

[Smith85] A. Smith, "Disk Cache – Miss Ratio Analysis and Design Considerations", *ACM Transactions on Computer Systems*, Vol. 3, No. 3, August 1985, pp. 161–203.

[Solworth90] J. Solworth, C. Orji, "Write-Only Disk Caches", *ACM SIGMOD International Conference on Management of Data*, May 1992, pp. 123–132.

[Solworth91] J. Solworth, C. Orji, "Distorted mirrors", *International Conference on Parallel and Distributed Information Systems*, December 1991, pp. 10–17.

[Stodolsky93] D. Stodolsky, G. Gibson, M. Holland, "Parity Logging Overcoming the Small Write Problem in Redundant Disk Arrays", *IEEE International Symposium on Computer Architecture*, May 1993, pp. 64–75.

[Teorey72] T. Teorey, T. Pinkerton, "A Comparative Analysis of Disk Scheduling Policies", *Communications of the ACM*, Vol. 15, No. 3, March 1972, pp. 177–184.

[Thekkath94] C. Thekkath, J. Wilkes, E. Lazowska, "Techniques for File System Simulation", *Software – Practice and Experience*, Vol. 24, No. 11, November 1994, pp. 981–999.

[TPCB90] Transaction Processing Performance Council, "TPC Benchmark B, Standard Specification", Draft 4.1, August 23, 1990.

[Treiber94] K. Treiber, J. Menon, "Simulation Study of Cached RAID 5 Designs", IBM Research Report RJ 9823, May 23, 1994.

[Vongsathorn90] P. Vongsathorn, S. Carson, "A System for Adaptive Disk Rearrangement", *Software – Practice and Experience*, Vol. 20, No. 3, March 1990, pp. 225–242.

[Wilhelm76] N. Wilhelm, "An Anomoly in Disk Scheduling: A Comparison of FCFS and SSTF Seek Scheduling Using an Empirical Model for Disk Accesses", *Communications of the ACM*, Vol. 19, No. 1, January 1976, pp. 13–17.

[Wolf89] J. Wolf, "The Placement Optimization Program: A Practical Solution to the Disk File Assignment Problem", *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1989, pp. 1–10.

[Worthington94] B. Worthington, G. Ganger, Y. Patt, "Scheduling Algorithms for Modern Disk Drives", *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1994, pp. 241–251.

[Worthington94a] B. Worthington, G. Ganger, Y. Patt, "Scheduling for Modern Disk Drives and Non-Random Workloads", Report CSE-TR-194-94, University of Michigan, Ann Arbor, March 1994.

[Worthington95] B. Worthington, G. Ganger, Y. Patt, J. Wilkes, "On-Line Extraction of SCSI Disk Drive Parameters", *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1995, pp. 146–156.

[Worthington95a] B. Worthington, "Aggressive Centralized and Distributed Scheduling of Disk Requests", Ph.D. Dissertation, University of Michigan, Ann Arbor, 1995.