SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 24(11), 981–999 (November 1994)

Techniques for File System Simulation

CHANDRAMOHAN A. THEKKATH
Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195. U.S.A.
(thekkath@cs.washington.edu)

JOHN WILKES
Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304. U.S.A.
(wilkes@hpl.hp.com)

AND

EDWARD D. LAZOWSKA
Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195. U.S.A.
(lazowska@cs.washington.edu)

SUMMARY

Careful simulation-based evaluation plays an important role in the design of file and disk systems. We describe here a particular approach to such evaluations that combines techniques in workload synthesis, file system modeling, and detailed disk behavior modeling. Together, these make feasible the detailed simulation of I/O hardware and file system software. In particular, using the techniques described here is likely to make comparative file system studies more accurate.

In addition to these specific contributions, the paper makes two broader points. First, it argues that detailed models are appropriate and necessary in many cases. Second, it demonstrates that detailed models need not be difficult or time consuming to construct or execute.

KEY WORDS Simulation File systems Performance

INTRODUCTION

Because file and disk systems are such critical components of modern computer systems, understanding and improving their performance is of great importance. Several techniques can be used to assess the performance of these systems, including abstract performance models, functional simulations, and measurements of a complete implementation. All have their place; we concentrate here on the use of simulations for detailed 'What if?' performance studies.

The main advantage of using abstract models as opposed to detailed simulation is their ability to provide adequate answers to many performance questions without the need to represent a great deal of system detail. However, abstract models make simplifying assumptions about aspects of the system that may be important, particularly in the later stages of a study when the design space has been narrowed and subtle issues are being considered. The work we describe here came about when we were doing some design studies of the interactions between current file system designs and new disk systems. In this environment, direct measurement was of

CCC 0038–0644/94/110981–19
©1994 by John Wiley & Sons, Ltd.

Received 1 October 1993
Revised 22 June 1994

course not possible, and we felt that an analytic model would be inadequate for our needs, at least partly because it would have difficulty representing the interactions between performance non-linearities in the disk system and the file system layout and request-sequencing policies.

In this paper we argue that detailed simulation studies for file and disk systems are often appropriate, and need not be difficult to perform. This paper describes a specific approach that can be used for such detailed simulations. This approach simplifies their execution, broadens their applicability, and increases the accuracy of their results.

The next section motivates the rest of the paper by describing a few earlier file system studies to show why it might be appropriate to use some of our techniques. The section following that briefly introduces the elements of our approach, which are elaborated upon in subsequent sections. We then present an application of the approach to a particular problem and summarize the key contributions of the paper.

MOTIVATION

As processors, memories, and networks continue to speed up relative to secondary storage, file and disk systems have increasingly become the focus of attention. The Berkeley Log-structured File System (LFS),[1] Redundant Arrays of Independent Disks (RAID),[2] and log-based fault tolerant systems[3,4] are some well-known examples of the newer innovative designs. Analysis of these systems has exposed many subtleties that affect performance. Current technology trends lead us to believe that file system and disk system design and analysis will continue to be one of *the* key areas in computer system design.

Typical performance studies of file systems involve the control of three distinct but related aspects: the disk, the file system, and the workload. In each of these areas, simple models trade off accuracy for modeling ease or tractability. Although useful early results can come from less detailed models with modest effort, these are no longer sufficient when more careful comparisons are desired. Indeed, back-of-the-envelope calculations or simple modeling of the software and/or the disk hardware can yield results that are contrary to real-life performance. We cite below some cases in point, where lack of detail or accuracy in the models led to predictions that turned out to be at variance with actual performance.

The selection of a rebuild policy for a RAID disk array is one example of the need for detailed and complete disk models. Here, a simulation study[5] using a detailed disk model that included rotation timing effects produced results that were contrary to an earlier study[6] that did not.

Another analytic model of a RAID controller[7] found that neglected factors such as contention within the various elements in the array controller caused actual performance to be noticeably worse than that predicted. Even simulations are not immune from over-simplification: a recent study on disk caching behavior found that ignoring the effects of read-ahead and write-behind in a disk model can produce results that are as much as a factor of two off from the actual disk.[8] Unfortunately, these and other real-life details are often omitted from models for the sake of simplicity. In a later section, we describe our disk model and suggest that neither accuracy nor modeling ease need be sacrificed.

The Log-structured File System is a file system whose performance characterization has evolved as more detailed models and simulations have been developed. We use it here as an example of how this process works. The earliest study[9] predicted a ten-fold improvement in performance based on a simple model that was based on micro-benchmarks. A subsequent study[10] using synthetic workloads provided improved accuracy, but over-estimated the cost of the segment cleaner by comparison with later measurements using a more realistic workload.[1]

Another group[11] looked at the effects of long running writes on read performance, an effect that had previously not been analyzed in detail. Finally, a more careful comparison of LFS and an improved regular file system that took cleaner costs into consideration found areas where each file system was superior to the other.[12]

The point here is not to criticize log-structured file systems or simple models, but to demonstrate that increasing the level of detail in the models used to study a file/disk system often reveals previously-hidden behaviors. This is normal, and to be expected; the purpose of this paper is to present techniques that will speed this process along.

SIMULATOR OVERVIEW

This section introduces the components that comprise our simulation environment for file and disk systems. Our approach is to derive workloads from I/O traces gathered from real systems, and feed these into real file system code sitting on top of a detailed disk model that has been calibrated against real disks. Additional software, which we refer to as *scaffolding*, holds all this together. Figure 1 shows how these components interact.

We use the term ‘simulator’ in this paper to refer collectively to all the components shown on the right side of Figure 1, while we use the term ‘file system simulator’ to refer to the component that mimics just the behavior of the file system code.

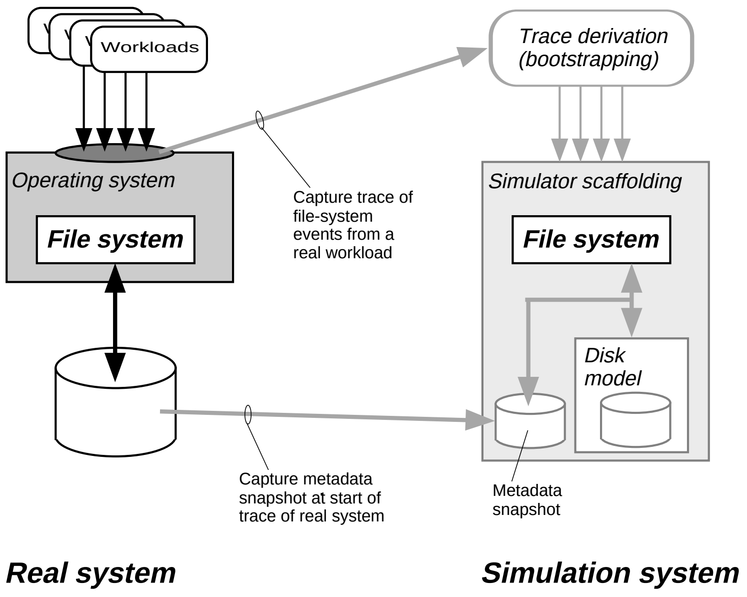
Workloads
Trace derivation (bootstrapping)
Operating system
File system
Capture trace of file-system events from a real workload
Simulator scaffolding
File system
Disk model
Capture metadata snapshot at start of trace of real system
Metadata snapshot
Real system
Simulation system

Figure 1. Simulator framework

As Figure 1 suggests, we first gather traces from the real system running a real workload. At beginning of the trace period, we take a snapshot of the file system metadata on the disk or disks being studied. This is the information kept by the file system to map <file, offset> pairs into disk block addresses. The trace is optionally used to derive a set of additional traces. Trace requests are supplied to the simulator. Within the simulator, the scaffolding component replays these to create a workload for the file system simulator. The file system simulator, in turn, emits requests to the disk model, which usually just performs timing calculations. In addition, if the request is a read to the metadata, the scaffolding intercepts the request and satisfies it from the previously-recorded snapshot; a metadata write is used to update the snapshot.

The remainder of this section provides an overview of each simulator component; they are discussed in greater detail in the subsequent three sections.

Workload traces

The accuracy of any performance study depends on both the quality of the model and on the quality of the workload representation that is used. The two usual sources for simulation studies are traces and synthetic workload models.

Synthetic workloads are considered more flexible than traces, and do not require significant storage because they are generated on the fly. However, in order to be realistic, synthetic workload models tend to be elaborate, difficult to parameterize, and specific to a single environment. One sample of a synthetic NFS-workload generator[13] uses 24 parameters to describe the workload—a wealth of detail that is not easy to gather.

Instead, our approach is to use trace-driven workloads, but to extend their utility through a technique known as *bootstrapping*, which is described further in the next section. This allows us to collect a single set of traces, and to generate additional sets while retaining certain statistical guarantees with respect to the original.

For investigating file systems that do caching, the most useful results are obtained by tracing requests at the system-call level: byte-aligned reads and writes, plus various control calls, such as file open and close, change directory, and so on.

We did our work on the HP-UX operating system, a POSIX-compliant Unix* system that runs on HP 9000 PA-RISC series 800 and series 700 systems.[14] The HP-UX system has a built-in measurement facility that can be used selectively to trace system events; several other operating systems have similar facilities, or one can be added relatively easily given access to the system's source code.

For our case study, we asked the kernel measurement system to gather information about all file-system related system calls, *fork* and *exit* system calls, and context switches. Together, these allowed our simulation scaffold to replay essentially exactly the sequence of events that took place in the original system.

There are a few important attributes of such trace-gathering systems for work of this kind: the traces must be complete (no records must be missed), they must be accurate (not contain invalid data), they must have precise timestamps (resolution of a few microseconds is acceptable), and gathering them must not disturb the system under test very much. The HP-UX trace facility met all these needs well: its timestamp resolution is 1 μsecond, and the running time for the tests we conducted increased by less than 5%—at least partly because we did trace compaction and analysis off-line.

The next section describes one of the contributions of this paper: a method called 'bootstrapping' for on-the-fly generation of additional traces from a previously collected set of traces. However, from the point of view of the simulator itself, each derived trace is handled the same way, so we will defer further discussion of this aspect for now.

* Unix is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Metadata snapshot

Before tracing is begun, a snapshot is made of the metadata on each of the file systems used by the workload being traced. This snapshot includes a copy of the file systems' naming hierarchy, i.e., the directories, overall size information, and a copy of the layout information. In our case, the HP-UX file system uses a slightly modified version of the original 4.2BSD Fast File System,[15] so this data included inode and cylinder-group maps.

The metadata snapshot is a copy of the data needed by the file system itself. The snapshot allows the scaffolding to provide the file system code under test the same data that it would have had access to, had it been running on the real system. In particular, the file system reads the directory data to do name lookups, and uses the layout information to turn user-level reads and writes into disk operations. As the simulation progresses, the file system under test modifies the metadata as a result of trace-driven user-level requests, and the scaffolding faithfully performs the requested updates so that future requests to the metadata will return the correct data.

Since the snapshot contains no data files, it is of modest size: a few percent of the total disk space being simulated. This is possible because we do not simulate the contents of user-data blocks, just their movement. Accesses to the metadata snapshot by the simulator scaffolding go though the real file system of the machine used to run the simulation, and so are subject to caching, which improves the elapsed simulation time.

Scaffolding

The scaffolding is the glue that binds together the entire simulation. It provides the following facilities:

1. Lightweight threads, i.e., execution contexts, which are used to simulate processes making file system calls and the concurrent execution of activities inside our disk simulator.
2. Time-advance and other mechanisms needed for the discrete-event simulation being performed.
3. Emulation of the kernel procedures that are accessed by the real file system. For example, *sleep* and *wakeup* calls are mapped onto synchronization primitives derived from the underlying lightweight thread library.
4. Software to access the metadata snapshot when requested by the file system code.
5. Software to manage the correct replay of an input trace.

At a high level, the working of the scaffolding is quite straightforward. The scaffolding uses one lightweight thread to simulate each independent process encountered in the trace. It then reads trace records that are fed to threads simulating user processes until the trace is exhausted, or the simulation has reached a sufficiently stable state that the desired confidence intervals have been achieved.

Each trace record is handed to the thread that is emulating the appropriate process. Most of these requests are read or write operations, which turn into calls on the file system code, and perhaps generate one or more simulated disk requests. Whenever the real process would have spent real time—e.g., while waiting for a disk access to complete—the thread is blocked, and waits for simulated time to advance to the appropriate point before it is allowed to proceed again. Once the request has been completed the thread goes back to wait for the next request for it to simulate.

If the operation being simulated is a *fork*, a new thread is created and associated with the child process, which then proceeds to accept and process requests, while the parent continues. Asynchronous disk requests do not cause the thread to delay.

Occasionally, the metadata snapshot needs to be consulted, and real data needs to be transferred between the snapshot and the buffer cache used by the file system code being run in the simulation. This is usually the result of a directory lookup or inode update. Note that the file system only invokes this mechanism when the needed metadata is not already in the simulated buffer cache. As a result, most reads and writes only do simulated data movement.

Disk simulator

Since we were interested in exploring the interaction between file systems and future disk designs, we chose to construct a disk model that could easily be tuned to reflect design changes extrapolated from current performance characteristics. We took some pains to calibrate this model against current real disks, and in particular, to include the effects of caching in the disk drive, which prior work had shown to be an important part of getting good agreement between a model and reality.[8,16] As a result, we were able to achieve differences between the real disk and the modeled one, i.e., the model demerit figure, of only 5%.[8]

The first component of our disk model is a buffer cache, which is used to keep track of data that has been read, read-ahead, or written. Appropriate replacement policies allow us to alter the behavior of this cache for different experiments. In addition, we modeled the physical disk mechanism—the rotating media and the moving disk head and arm—and the DMA engine used to transfer data from the disk cache to and from the disk-to-host bus. By making each of these lightweight tasks, we were able to model the overlap between disk accesses and data transfers to and from the host system that occurs in real disks.

Our model uses replaceable modules for each of these components, so it is relatively easy to make changes to explore different design choices. For example, enhancing the disk model to predict the effects of making its buffer cache non-volatile, to be described in the case study, took less than a day.

File system simulator

The other important component we were interested in modeling was the file system. One approach to modeling a file system is to construct a simplified simulation of the file system code. By making suitable assumptions, the resulting complexity and development time could be kept within reasonable bounds. However, this is a process fraught with difficulties, as our examination of the LFS development process suggests. Ensuring that the right simplifications are made is difficult, doubly so because the expectations of the experimenter can often bias the choices in favor of the set of assumptions made during the design of the file system that is being investigated.

We believe that it is possible to do better. In fact, our approach is to use the real file system code instead of an imperfect model of it, thereby eliminating any possibility of incorrect assumptions. To do this, we bring the file system out of its normal execution environment, which is the operating system kernel or a trusted address space. We do this by providing a set of scaffolding that looks—as far as the file system is concerned—just like the kernel environment in which it normally runs, down to and including the synchronization primitives the code is written to invoke. The entire ensemble runs as a regular, untrusted, user-level application.

File system designers have used this technique before to run kernel-level code at user-level to simplify debugging during program development,[15] but not, to our knowledge, explicitly for performance studies.

Our approach allows the file system implementation and an understanding of its performance to develop together. For example, in addition to providing functionality stubs for incomplete portions of the code, we can provide performance stubs as well. Running the new design in a user-space scaffolding, as in our approach, combines the advantages of easier development, faster turnaround time, and more flexible debugging with early access to performance data.

In the case study we conducted, we used the production HP-UX file system code as the file system simulator.

Analysis: detail and complexity versus efficiency

We argue here for the use of detailed models for file system and disk system components. Our contention is that such models lead to increased confidence and increased accuracy, without an excessive increase in execution time or complexity.

Consider first the degree of detail that is *desired* in modeling the system. Obviously, if the real system is available to test, it is usually best to measure that system, since this minimizes the uncertainty. One of the strengths of our approach is the use of real file system code as the file system simulator, which removes one major cause of uncertainty; another is the use of real traces rather than synthetic ones; a third is the calibration of the disk system against real disks.

Consider next the degree of detail that is *required* to model the system. Work in disk drive modeling has shown that detailed models are a necessity there: ignoring caching effects, which in turn depends on modeling rotation position in the disk, can result in mean simulated times as much as a factor of two larger than they should be.[8] So, sufficient detail is essential if useful results are to be acquired.

Of course, it is not always possible to determine which features of the model will prove to be the most important—indeed, these may change as a function of what is being modeled. For example, a file system that did not make rotational-position layout optimizations or use the disk's aggressive write caching would be much less sensitive to caching effects in the disk. Thus, we believe it prudent to err on the side of caution.

Finally, consider the *cost* of detailed models. We believe that the approach we advocate is not particularly costly: our disk simulator is able to process about 2000 requests per second on a 100 MHz PA-RISC processor; the file system code runs at full processor speed; and—just as in real life—the metadata snapshot information is frequently cached by the underlying real file system that the simulator is hosted on. The result is that the elapsed time for executing the simulations is much less than that required to execute the real system executing the traced workload. Furthermore, as processors speed up relative to I/O, this disparity in performance is likely to increase.

A significant benefit of our approach is *confidence* in the results. In the final stages of design, omitting a crucial detail may be potentially dangerous. Our approach makes it easy to construct a detailed model that avoids this pitfall.

We feel the accuracy and confidence offered by our approach far outweigh the small investment in time to build the scaffolding. This is a one-time cost that can be amortized over many studies. In our experience, the code to implement a detailed disk model and the bootstrap generator proved fairly straightforward. By simply dropping the real file system code into the simulator, our development time for this portion of our model was zero.

On the other hand, a potential drawback to our scheme is that it assumes the availability of the file system code. Sometimes this might not be case, e.g., when designing a new file system from scratch. However, even in these cases, many aspects of the our system, e.g., the disk model, workload characterization, and parts of the scaffolding, may be used independently.

CONSTRUCTING A WORKLOAD

As mentioned in the introductory section, traces and synthetic workloads each have advantages and disadvantages. Traces are more realistic but tend to be very voluminous. Synthetic workloads offer flexibility at the loss of verisimilitude. We use a technique that combines the good elements from both approaches.

Our method is derived from a statistical technique called *bootstrapping*,[17] which can be used to increase the confidence and reliability of scientific inferences. The basic idea is as follows. Given a sample of size N of some population with an unknown distribution, we generate some number of new samples of size N by selecting elements at random from the original sample. The elements are selected *with replacement*, which means that copies of the same element can occur more than once in the new sample. Note that we can generate N^N different samples of size N.

For each sample, called a *bootstrap*, assume we calculate some sample statistic, say, the average. The calculated statistic from each of the bootstraps constitutes a distribution, called the ‘bootstrap distribution’. Bootstrap theory says that the bootstrap distribution can be treated as if it were a distribution created from samples drawn from the *real* population. Thus, it can be used to estimate the accuracy of the statistic, in this case the average, that was calculated from the original sample.

A useful pragmatic aspect of bootstrapping is that a new sample of size N can be created on-the-fly. This can be done by numbering the elements in the original sample sequentially from 1 to N; generating N random numbers in the range; and then selecting element i to be in the bootstrap whenever the random number generator comes up with the number i.

Bootstrapping is a well-established technique in statistics that has not, to our knowledge, been used in computer systems analysis. The technique can be applied to file system traces in a straightforward way to generate many trace sets given a single trace. To a first order, we use the individual processes that show up in our traces as the ‘elements’ for the bootstrapping process. The truth is slightly more complicated, and elaborated on below. Selection of a process element implies inclusion of all the I/Os it issued in the resulting bootstrap trace.

We generate bootstraps—as many as required for the experiment; the number available is effectively unlimited for any realistic value of N—and run it against the simulator. With high probability, the bootstrap distribution has statistical properties that are similar to the real-life population. Thus the behavior of a system when run against the bootstraps is likely to approximate its behavior in real life. Note that bootstrap theory does not imply that the behavior of the simulator on *individual* bootstraps will be be the same as the original trace. However, the average behavior of the simulation on a set of bootstraps approximates the average behavior of the simulator if it were run on a set of real traces.

Independence of sample elements

In bootstrapping theory, the elements of a sample are assumed to be independent of each other. Consequently, the degree of dependence between sample elements in a bootstrap will affect the final results. This affects practical file system studies in two ways.

There are two kinds of inter-dependence that matter: *functional dependence*, e.g., a file system will not allow write operations on a file that has not been opened, and *behavioral dependence*, e.g., if process P1 writes once to the file, process P2 will read six things from it. We need to be concerned with both; bootstrapping theory is strictly only concerned with the latter.

Functional dependence requires that the elements used in the bootstrap be large enough to include the necessary system state. For example, an element that includes a read from a file must also include the system call that opened that file. In practice, when a file-system study of the type we conducted is being done, this means that elements are at least as large as processes, and may sometimes have to be groups of processes forked by a common ancestor if the ancestor performs file open operations that the child processes rely on.

If processes exhibit behavioral dependence, e.g., they communicate using pipes or synchronize on a common file, it is necessary to aggregate these processes into larger units that are then treated as independent sample elements.

Together, these adjustments can decrease the number of elements for the bootstrapping. However, it is important to note that the quality of input to the simulator is not degraded. First, even with a modest number of elements in a sample, there is no practical limit to the number of bootstraps that can generated. For example, even if N is only 5, 3125 ($= 5^5$) possible bootstraps can be generated. Second, since we are only aggregating the amount of file system activity, and not discarding samples, each bootstrap continues to offer roughly the same load to the file system as the original trace. This is true even if only a single bootstrap can be built because all the input has to be aggregated into one element.

Implementation and validation

In our implementation, bootstrap generation is done as a three stage software pipeline. The first stage rolls a die multiple times to choose a set of processes that are to be included in the bootstrap. The second stage deletes the traces of the processes that are not part of the bootstrap. The final stage duplicates the traces of processes as necessary; new process identifiers and sequence numbers are created at this stage. Recall that creating a bootstrap involves selection with replacement. This process allows the individual elements to execute independently of one another.

Duplicated records have the same time-stamps as the original records they are derived from. This could lead to increased contention for file and disk resources. In our experiments, this has not been a significant issue for the average case behavior because of the filtering performed by the user-level file cache in the file system simulation.

For a given trace, we generate multiple bootstraps and run the simulator on each bootstrap, and then aggregate the performance data that results across these runs. Bootstrapping theory tells us that bootstrap distribution of a particular statistic closely approximates the true, but not directly measurable, distribution of the statistic in the real population. Thus on the average, the performance of the simulator on the bootstraps will be similar to what would have been seen if it had been run on real traces, i.e., samples from the real population.

The entire process of generating bootstraps can be done on-the-fly. It is also repeatable, if the same pseudo-random number generator is used for the selection process. This means that exactly the same bootstraps can be generated several times if so desired, e.g., for runs with different simulation parameters.

The main value of using bootstrapping in simulation studies is to extend existing trace data on-the-fly while retaining certain statistical guarantees. As long as the original trace data was a representative sample of the original system, bootstrapping theory tells us that the aggregate of the bootstraps will also be representative of the real system. That is, rather than collecting and storing multiple traces, on-the-fly bootstrapping allows you to achieve the same effect as if multiple traces were collected—but with much less effort.

MODELING THE DISK

Although there are some exceptions, much of the prior work on disk modeling has not accurately reflected the considerable concurrency that occurs in modern disk drives, nor the actual operational characteristics of the disk itself, including non-linear seek versus distance times, bus transfer effects, and caching. We endeavored to address these issues in our model. Its calibration has been described elsewhere;[8] here, we concentrate on a description of the elements that go to make up the model.

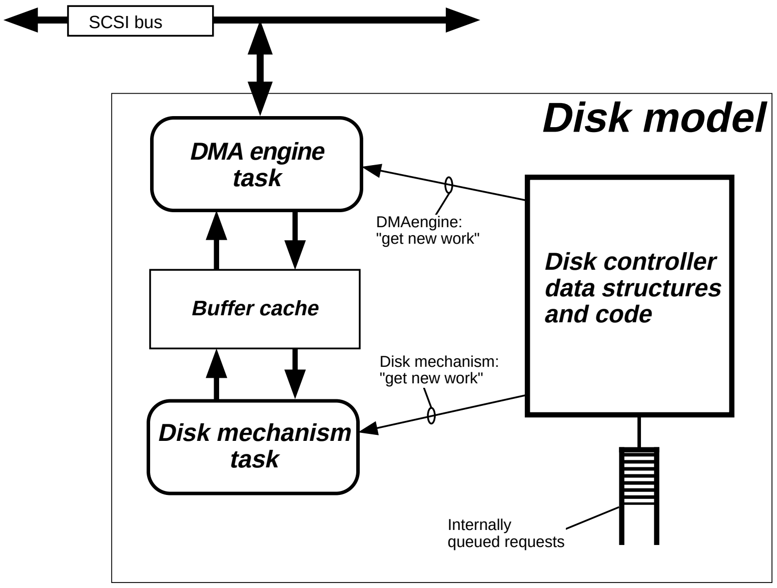
SCSI bus
Disk model
DMA engine task
DMAengine: "get new work"
Disk controller data structures and code
Buffer cache
Disk mechanism: "get new work"
Disk mechanism task
Internally queued requests

Figure 2. Major components of the disk model

Figure 2 shows the components of our model. The major components are as follows; in our implementation, each is a C++ object:

1. The *disk controller* is a data structure that binds together the other elements, and provides a placeholder for them. It also provides the management code for the disk's actions, and a number of parameters used to control modeling of aspects like controller overhead.
2. The *disk cache* represents the on-board cache memory in the disk. It can be managed as a simple speed-matching buffer, or segmented and used to cache data before or after it is explicitly referred to by the host. Here are two examples of the caching policies that our model supports. If the disk is idle, and the last request was a read, the controller may choose to continue doing a speculative read-ahead into the cache in case the host is making sequential transfers. If the last request is a write, the controller may allow data

transfer across the bus into the disk in parallel with the execution of the last request; this is known as immediate reporting, and allows efficient writes to consecutive disk addresses.

3. The *disk mechanism* task models the rotating media and the disk heads attached to a moving arm. In practice, most of the code is concerned with translating logical addresses into physical ones, taking into account details of the disk drive geometry such as zoning, which allows more sectors on the outer tracks than the inner ones spare sectors, and head- and track-sector skew, which minimize rotation delays on head and track-switches.
4. The *DMA engine* task models the transfer of data across the interface between the disk and the bus connecting it to the host system. The bus is acquired and released according to policies determined by the design of the disk controller, parameters that can be set by the host system, and the availability of data or space in the disk cache. This allows contention between multiple disks on the same bus to be modeled correctly.
5. The *request-scheduling policy* determines, in combination with the cache-management policies, which request will be executed next if the disk drive has been passed more than one. For example, this allows the command queueing of SCSI-2 to be modeled.

We found it convenient to have each task call into the disk controller code to request work for it to do, blocking if there was none. This allowed each task to be a simple get-work—execute-it loop, and let us concentrate the complexities of handling the interactions between the cache management and the request scheduling in one place.

While this model might appear complex, it is in fact quite easy to implement. Our scaffolding provides lightweight threads, synchronization objects such as semaphores, and queue abstractions. The disk elements are implemented as independent threads that send messages to each other through queues and synchronize as needed using semaphores. The model has been parameterized for several different disks using a combination of manufacturer-supplied data and direct measurements. The simulation is tuned to minimize error in the transfer size range, typically 4–8 kbytes, commonly used by current file system designs. Calibration against real disk performance under a range of workloads yields excellent agreement, within 5%. The total code required to achieve this level of accuracy is modest—a little over 3000 lines of C++.

The particular disk model that we describe here has been extensively used in other studies. A separate paper[8] contains quantitative information of how different portions of the model contribute to its accuracy and how it compares with typical simple models. Undoubtedly, an accurate model like ours is more complicated than a simpler, less accurate, model. On the other hand, we can quantify and bound its deviations from the behavior of real disks, and we know that it does a good job of modeling components of disk behavior that are growing in importance as file system designs attempt to adapt to, and take advantage, of exactly these performance non-linearities.

MODELING THE FILE SYSTEM

Since we were particularly interested in exploring the effects of changing disk technology on file system behavior and performance, we developed techniques that allowed us to use the actual file system code rather than an imperfect abstract model of it. By comparison with an abstract model, our approach increased our confidence in the results, and also ensured that we did not have to continually adjust the parameters of the file system model as a result of different workloads or disk behaviors.

We found it straightforward to adapt the file system code running in the kernel to run as an untrusted user application within the simulator. The infrastructure requirements of a file system are typically straightforward: some multitasking, simple memory management, and access to physical devices and user memory space—usually through a very stylized, well-controlled interface. The multitasking support usually has to include some form of threads and a set of synchronization primitives. All these are relatively easy to emulate in a user-space scaffolding. For example, the device-driver routines can easily be provided by a set of procedures that invoke the interface provided by the disk simulator. Processes in the original system can be treated as independent threads each with per address space structures imitating those of the original system. Though we happened to use the HP-UX file system as a base for our case study, these techniques are applicable in exporting code from other systems to run at user level.

As a specific example, for the case study to be described later, the entire HP-UX file system,[14] which is derived from the 4.2BSD Fast File System,[15] was run at user level without modification. In this case study, almost all the code in the file system simulator was taken from a copy of the HP-UX product source code. Additional code that was needed to make it execute correctly at user level was quite minimal—about 3000 lines of C. This represents code that is implemented once; the actual code for the various file systems under test runs unchanged. This represents a huge saving in work, because typical file system implementations are quite large. Most of the code we added is required to provide the right kernel-level abstractions and the correct device interface at user level and can be reused without any change to simulate other file systems.

To validate our file system simulator implementation against a real kernel, we compared the block requests issued by the real file system running inside the operating system kernel and the simulator. There were no significant differences between the two systems on a set of several different programs. This is not too surprising: we were executing the same code in both cases, but we found that it inspired our confidence in our results.

SCAFFOLDING

The scaffolding is perhaps the most important piece of the simulator framework. Apart from providing the basic framework for a discrete-event simulator, it binds together the various pieces of the simulator and provides interfaces that are appropriate to each. The preceding sections have touched upon the facilities provided by the scaffolding to the file system simulator and the disk simulator. This section expands on those, and describes some of the details related to the discrete-event simulation.

Threads

The central element of the scaffolding is its coroutine or threads package. We happened to pick one built upon the base provided by a standard, off-the-shelf library from AT&T,[18] modified slightly to support time calculations using double-precision floating point instead of integer arithmetic. Our choice of the thread library was dictated by what was most conveniently available to us. Other packages such as PRESTO[19] could probably also be used with minimal modifications. Such coroutine packages typically provide a set of objects including lightweight threads, synchronization objects, and communication channels such as queues. They need not implement preemptive multitasking, but should at least support the notion of simulated time, in which a thread can delay for a while to represent passage of real time, and then be resumed once simulated time has advanced sufficiently.

Trace replay

The trace records we used contained file system calls made by programs, in addition to records for process creation and deletion. Each trace record contained the process identifier of the caller, parameters to the system call, and two time-stamps—one indicating when the system call was initiated, and the other when it completed.

For trace records that indicate a process creation, the scaffolding creates a new thread, complete with whatever kernel level state that the original process had. For instance, in the Unix model, open file descriptors and the current working directory have to be inherited from the parent process.

After each system call is performed, the thread goes back to waiting for more work. When a process deletion record is encountered, the thread emits some statistics, and is then terminated.

For trace records that denote normal file system events, the scaffolding hands off the request to the previously-created thread associated with the process that issued the request. The thread then delays for a while to represent application compute- or think-time. This period is the interval between completion of the last real request and this one, as recorded in the trace. A faster CPU can be conveniently modeled by decreasing the duration of this pause. Notice that the simulated time at which the new request is issued may not be the same as the real time recorded in the trace: the file system and disk subsystem models may execute the request in more or less simulated time as a result of many factors, including design changes.

After the delay has expired, the thread executes the system call described in the trace record—typically by calling into the file system, which executes its code exactly as if it had been invoked by a process executing in the kernel.

When the file system call returns, it is necessary to advance the *simulation time* by the amount of *real time* it would have taken to execute the file system code. A convenient way to do this is to measure the time taken to execute the file system simulation code, which is, after all, the real code, and then reduce this, if necessary, by a constant factor representing the difference in CPU speeds of the processor being modeled and the one on which the simulation is running.

As part of executing the file system code, disk requests that might be made are also charged against the thread. Notice that other requests can enter the file system code, subject only to the synchronization constraints imposed by the scaffolding, which is in turn a faithful model of the real system's rules. This usually allows concurrent outstanding requests from different processes to be executing inside the file system. If the scaffolding failed to support this, it would underestimate the effect of contention in the file and disk system and could lead to erroneous results.

As a practical matter, maintaining the appropriate level of concurrency in the simulation was one of the thorniest issues that we had to deal with. We got it wrong several times, eventually settling on the following scheme.

Each thread has a private input queue of requests, i.e., trace records, that it is expected to process, sorted by their start time. When a thread finishes its current request, it attempts to get the next record to execute from this input queue. If there is something there, it executes it. If not, and the thread has not yet been terminated, the scaffolding reads down the trace input until it finds the next request for this thread. Since the input trace is sorted by completion time of the original system calls, not by process number, it is quite likely to read several records before it finds one for this thread. These records are appended to the ends of the work queues for the relevant threads.

This scheme could potentially require reading ahead arbitrary amounts of trace input:

consider a thread that does a 'sleep forever'. Thus, the degree of read-ahead has to be controlled in each simulation. This limit is set high enough, typically a few thousand requests, so that the amount of lost concurrency is very small, and the performance effects negligible.

Correct sequencing of process creation and deletion is achieved by having the threads themselves perform the *fork* and *exit* calls that alter the scaffolding state.

File system support

Our scaffolding provides the necessary support environment for the kernel-level file system used in HP-UX to run in user mode. It supports the notion of the UNIX system's per-process *u-area*, provides support for *sleep* and *wakeup* synchronization calls, and emulates the kernel trap/return mechanism.

Some aspects of the original kernel implementation are simplified in the simulator. For instance, the kernel-level routines used to transfer data safely between user and kernel spaces, as well as the memory allocator, have been simplified without affecting their interfaces. We were also able to simplify interrupt handling by eliminating the kernel mechanism for vectoring and dispatching the interrupt to the device driver.

The file system code is presented with the same disk interface that it would see in the kernel. However, when a call is made to read or write a disk, the scaffolding intercepts it, and passes it on to a disk simulator read/write routine. The disk simulator routines neither produce nor consume any real data. Data from the calling process is usually discarded unless it is metadata that might be needed later. In this case, it is added to the snapshot data by invoking real system calls. Similarly, data given back to a process is usually meaningless, unless it is metadata or directory information that has to be used by the file system. In such cases, the snapshot is consulted for the data.

In our implementation, we did this check by comparing the block number issued by the file system against a lookup table hashed by metadata block numbers. This structure also provided a pointer to where in the metadata snapshot the block was stored. Simulated time was not advanced while the snapshot is consulted: as far as the file system could tell, the request took just as long as the disk timing model said it should.

CASE STUDY

The previous sections have described a collection of techniques for file system simulation. In this section we describe a case study that employs these techniques. Through our case study, we wish to demonstrate that:

1. Our disk model is detailed enough to study the effect of important and practical changes that would be difficult or impossible to study otherwise.
2. Bootstrapping is an efficient and useful technique and can be used instead of extensive trace generation or synthetic workload models.
3. Our scaffolding mechanism is powerful and robust enough to handle code from a real file system.

We demonstrate the first two of these goals directly through quantitative experiments. We demonstrate the last goal indirectly by running the production-quality HP-UX file system within the scaffolding.

We do not present detailed comparative measurements of file systems, e.g., FFS and LFS, because our goal is to demonstrate that the methodology works and is 'real', rather than to conduct an extensive and definitive performance analysis. Instead, we have chosen to present a simple study that underscores the use of the separate techniques rather than a large, complicated case study where it might be difficult to isolate the advantages of using each of our techniques.

Modification analysis

The baseline system we measured consists of the standard Fast File System from HP-UX version 8.0 running a multiprogramming benchmark designed to be typical of a program development environment. The benchmark is quite I/O intensive and includes program editing, compilation, and miscellaneous directory and file operations. The multiprogramming level was set to 20 users, to approximate the situation when 20 programmers are working. The files used by the benchmark were stored on a disk that is essentially identical to the model shown in Figure 2.

The modification analysis that we undertook is related to disk system design. We studied the performance changes in the original system when the track buffer was made non-volatile. This has the effect of speeding up writes, which now complete as soon as the data is in the buffer without waiting for the disk mechanism. Synchronous writes, which slow the performance of many file systems, are no longer on the critical path with this design. The write back policy used to clear the buffer to disk is similar to C-SCAN,[20,21] modified so that read requests take precedence over write requests so as not to introduce delay to user-level requests. We expected that introducing a non-volatile RAM into the disk would improve throughput and would allow the disk to keep pace with increased CPU speeds up to a point. This type of modification analysis is very useful in disk system design and allows cost/performance comparisons between hardware-based and software-based logging solutions to the problem of synchronous writes.

The significance of using our detailed disk model lies primarily in being able to undertake the modification analysis. Accounting for the effect of the track buffer and the writeback policy would be quite difficult with a more abstract model. In addition, the quantitative result of the analysis also has a high degree of confidence because of the fidelity of the disk model. Accuracy tends to be a significant factor as cost-performance margins narrow in the later stages of design.

Figure 3 graphs the peak throughput of the baseline system and the modified system, executing traces from the original sample.

Throughput is measured as the rate of data transfer between the file system and the disk. The X axis shows relative CPU speed, i.e., it models the performance effects of increased CPU speeds without any change in the disk speed. Predictably, the throughput levels off as CPU speed increases, because the bottleneck becomes the disk service time. The graph shows that saturation occurs when the CPU speed increases by about 100% over the baseline value.

The new organization has roughly 50% better overall throughput and saturates at about 50% higher CPU speed than the original. Thus, this approach would indeed be a feasible improvement for existing file systems if the hardware costs were reasonable.

Effectiveness of bootstrapping

Here we demonstrate the behavior of bootstraps compared to the original trace data. As mentioned earlier, the theory behind bootstraps guarantees certain statistical properties on the resulting data set. In particular, the distribution of a particular statistic got by running multiple

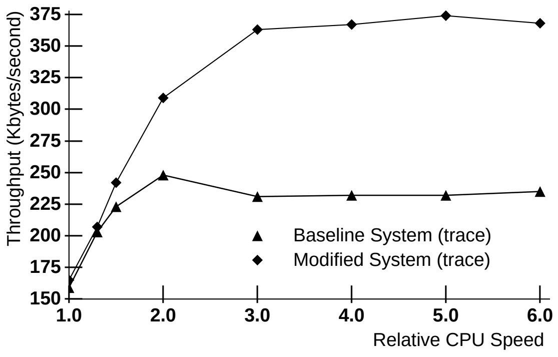

Figure 3. Comparison of the baseline disk system with one that adds an asynchronous write-back cache, using the original trace

bootstraps is as good as that generated by running different traces, i.e., samples, from the real population.

Each element in our sample is a process generated on behalf of a synthetic user in our 20-user benchmark run. Each of these processes is represented by several, possibly many, trace records; in turn, each synthetic user is represented by many processes generated on their behalf.

A complication that applies to this simple labeling of processes as elements is that some processes depend on the work done by others: for example, consider the situation where process P1 creates a directory, P2 reads files from the directory, and finally P3 deletes the directory. If a particular bootstrap included processes P2 and P3 but not P1, its execution would fail because it would not contain the 'create directory' trace record. Similar problems would arise if we took individual trace records as the element for the bootstrap.

We solved this problem by exploiting our knowledge of the benchmark. We identified interdependent processes, like P1, P2, and P3 above, and aggregated them into a single sample element. Because we understood the benchmark, we were able to do this manually; it would be straightforward to generate a tool that determined these interdependencies from the input trace, since it contains all the information needed.

As mentioned previously, aggregation does not affect our ability to generate large number of bootstraps or the number of requests placed on the simulator. Based on our experience with our benchmark, this aggregation has not been a serious problem. Further, if the elements of the trace were individual disk block accesses, such aggregation would not be required because we could treat the requests as independent from the viewpoint of the disk system being studied.

We performed a first-order evaluation of how well the bootstrapping technique generated workloads by comparing the simulated performance of the original baseline case from Figure 3 against several different bootstraps generated from it. The results are shown graphically in Figure 4 and in tabular form in Table I. In this case, the results are in good agreement to that produced from the real trace. Even if they happened not to be, recall that bootstrapping theory says that the *average* behavior on a large number of bootstraps is a good indicator of behavior on a large trace.

The exercise here was one that was designed to explore the use of bootstraps in practical simulations. In reality, the point of using bootstraps is not to repeat a given experiment but to

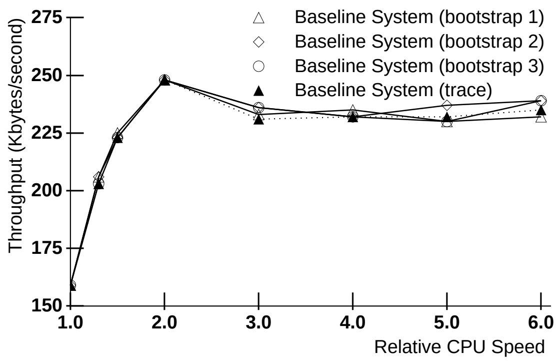

Figure 4. Simulation performance resulting from several sample bootstraps

extend the trace data. That is, rather than generate and store multiple traces, bootstrapping is used to extend the trace, and the results from these are used as indicators of the behavior of the system on a set of real traces.

CONCLUSIONS AND CONTRIBUTIONS

Technology trends suggest that the design of file and disk systems will continue to be a very important area. To this end, this paper has described a set of techniques that can be used to improve file and disk system performance studies by increasing both the accuracy and the efficiency of careful simulation studies.

The techniques presented in the paper fall into three areas: workload generation, file system modeling, and disk modeling. In each of these areas, the paper makes novel contributions, some of which are enumerated below:

1. The use of bootstrapping in creating traces with known statistical properties appears to be novel in the computer science community, even though it is a well-known approach in the statistics community.
2. The file system model and scaffolding demonstrate that by employing simple primitives and a modest amount of programming effort, it is possible to generate a faithful file system model that incorporates the detail and subtlety of the actual system under test. As a basis for comparison, we spent roughly 2–3 person weeks implementing this portion of the code.

Table I. Variations of the sample bootstraps from the original trace

	Percentage variation in throughput with CPU speedup							
Sample	1.0	2.0	3.0	4.0	5.0	6.0	Mean	Std. Dev.
Original	-	-	-	-	-	-	-	-
Bootstrap 1	0.0	0.0	0.9	1.3	-0.9	-1.2	0.0	1.0
Bootstrap 2	0.0	0.0	2.2	0.0	2.2	1.7	1.0	1.1
Bootstrap 3	0.0	0.1	2.2	0.2	0.0	1.7	0.7	1.0

3. Our use of the scaffolding for the file system code in a way that emphasizes obtaining performance information is an extension to prior work, which emphasizes code debugging.
4. Our detailed disk model shows that it is possible to capture the nuances of behavior of complex modern disk drives, while retaining good execution speed for the simulator.

Notice that each of the techniques we describe can be used independently of the others. For instance, bootstrapping can be used to generate workloads without using the file system scaffolding or the disk simulator.

We believe that detailed simulations and simple analytic models both have an important place in understanding the behavior of complex systems. The simple models are crucial in narrowing the design space to manageable proportions, at which stage additional detail is required for realistic evaluations. The approach and techniques we present here allow this detailed simulation step to be applied more readily, across a greater set of workloads, and with greater confidence in the results than would otherwise be the case.

ACKNOWLEDGEMENTS

Chris Ruemmler wrote the first versions of the disk simulator that this work was built on. Thanks are due to Radhika Thekkath for her patient readings of the paper and suggestions for improving it. We also wish to thank the anonymous referees for several helpful comments.

This work was supported in part by the National Science Foundation (Grants No. CCR-8907666, CDA-9123308, and CCR-9200832), the Washington Technology Center, Digital Equipment Corporation, Boeing Computer Services, Intel Corporation, Hewlett-Packard Company, and Apple Computer. C. Thekkath was supported in part by a fellowship from Intel Corporation.

REFERENCES

1 Mendel Rosenblum and John K. Ousterhout, ‘The design and implementation of a log-structured file system’, *ACM Transactions on Computer Systems*, **10**, (1), 26–52, (February 1992).

2 David Patterson, Garth Gibson, and Randy Katz, ‘A case for redundant arrays of inexpensive disks (RAID)’, *ACM SIGMOD 88*, 109–116, (June 1988).

3 David J. DeWitt, Randy H. Katz, Frank Olken, L.D. Shapiro, Mike R. Stonebraker, and David Wood, ‘Implementation techniques for main memory database systems’, *Proceedings of SIGMOD 1984*, June 1984, pp. 1–8.

4 Robert B. Hagmann, ‘A crash recovery scheme for a memory-resident database system’, *IEEE Transactions on Computers*, **35**, (9), 839–843, (September 1986).

5 Mark Holland and Garth A. Gibson, ‘Parity declustering for continuous operation in redundant disk arrays’, *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992, pp. 23–35.

6 Richard R. Muntz and John C.S. Lui, ‘Performance analysis of disk arrays under failure’, *Proceedings of the 16th Conference on Very Large Databases*, 1990, pp. 162–173.

7 Ann L. Chervenak and Randy H. Katz, ‘Performance of a disk array prototype’, *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1991, pp. 188–197.

8 Chris Ruemmler and John Wilkes, ‘An introduction to disk drive modeling’, *IEEE Computer*, **27**, (3), 17–28, (March 1994).

9 John K. Ousterhout and Fred Douglis, ‘Beating the I/O bottleneck: A case for log-structured file systems’, *Operating System Review*, **23**, (1), 11–27, (January 1989).

10 Mendel Rosenblum and John K. Ousterhout, ‘The LFS storage manager’, *Proceedings of the Summer 1990 USENIX Conference*, June 1990, pp. 315–324.

11 Scott Carson and Sanjeev Setia, 'Optimal write batch size in log-structured file systems', *Proceedings of the USENIX Workshop on File Systems*, May 1992, pp. 79–91.

12 Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin, 'An implementation of a log-structured file system for UNIX', *Proceedings of Winter 1993 USENIX*, January 1993, pp. 307–326.

13 Roberta A. Bodnarchuk and Richard B. Bunt, 'A synthetic workload model for a distributed system file server', *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1991, pp. 50–59.

14 Frederick W. Clegg, Gary Shiu-Fan Ho, Steven R. Kusmer, and John R. Sontag, 'The HP-UX operating system on HP Precision Architecture computers', *Hewlett-Packard Journal*, **37**, (12), 4–22, (December 1986).

15 Marshal Kirk McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry, 'A fast file system for UNIX', *ACM Transactions on Computer Systems*, **2**, (3), 181–197, (August 1984).

16 Chris Ruemmler and John Wilkes, 'Unix disk access patterns', *Proceedings of the Winter 1993 USENIX Conference*, January 1993, pp. 405–420.

17 Persi Diaconis and Bradley Efron, 'Computer-intensive methods in statistics', *Scientific American*, **248**, (5), 116–130, (May 1983).

18 AT&T, *Unix System V AT&T C++ language system release 2.0. Selected readings*, 1989.

19 Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy, 'PRESTO: A system for object-oriented parallel programming', *Software – Practice and Experience*, **18**, (8), 713–732, (August 1988).

20 P. H. Seaman, R. A. Lind, and T. L. Wilson, 'On teleprocessing system design: Part IV: An analysis of auxiliary-storage activity', *IBM Systems Journal*, **5**, (3), 158–170, (1966).

21 Robert Geist and Stephen Daniel, 'A continuum of disk scheduling algorithms', *ACM Transactions on Computer Systems*, **5**, (1), 77–92, (February 1987).