# A Detailed Simulation Model of the HP 97560 Disk Drive

David Kotz, Song Bac Toh, and Sriram Radhakrishnan

Department of Computer Science
Dartmouth College
Hanover, NH 03755-3510
dfk,songbac,sriram@cs.dartmouth.edu

Dartmouth PCS-TR94-220

July 18, 1994

**Abstract**

We implemented a detailed model of the HP 97560 disk drive, to replicate a model devised by Ruemmler and Wilkes (both of Hewlett-Packard, HP). Our model simulates one or more disk drives attached to one or more SCSI buses. The design is broken into three components: a test driver, the disk model itself, and the discrete-event simulation support. Thus, the disk model can be easily extracted and used in other simulation environments. We validated our model using traces obtained from HP, using the same "demerit" measure as Ruemmler and Wilkes. We obtained a demerit percentage of 3.9%, indicating that our model was extremely accurate. This paper describes our implementation, and is meant for those wishing to use our model, see our validation, or understand our code.

## 1 Introduction

A recent paper by Ruemmler and Wilkes [RW94] describes the HP 97560 disk drive in detail, their model of the disk, and general techniques for modeling disks. We implemented our own version of the model. Our model simulates one or more disk drives attached to one or more SCSI buses. The design is broken into three components: a test driver, the disk model itself, and the discrete-event simulation support. Thus, the disk model can be easily extracted and used in other simulation environments. We have used it in a stand-alone mode (which uses a small discrete-event support module that we built) and incorporated it into a larger simulation embedded in the Proteus parallel-architecture simulator [BDCW91].

This paper does not describe any new results; it is meant as documentation of our model, its implementation, and its validation. There are three sections in the paper, oriented to readers with

This document corresponds to version 2.0 of the disk-model software. This project was supported by research funds from Dartmouth College.

different interests:

**Usage:** Section 2 describes the interface seen by a "user" of the disk model, i.e., the driver program.

**Implementation:** Section 3 describes the HP 97560, our implementation of the disk model, and the support code needed to run it.

**Validation:** Section 4 describes our validation process and the results.

The sections should be readable independently.

## 2 Usage

This section describes the user interface for the disk device, i.e., the interface used by a driver program that generates requests for the disk (e.g., from a trace file, or from a simulated file system). More details are found in the code, in particular, in `diskdevice.h`. Essentially, the diskdevice module exports a set of access functions.

`DiskDeviceInit(disk, busId, busOwner, busWaitq, fileName)`

This routine is called once for each disk. Any number of disks may be defined, with any disk ID numbers (non-negative integers). `busId` is the number of the bus to which this disk is attached and is used for debugging. `busOwner` is a pointer to an integer (one per bus) that holds either the number of the disk currently using that bus, or `BUS_FREE`. Finally, `busWaitq` is a queue where disks can wait for service by that bus. Our model manages the bus, but this interface allows the caller to decide which disks share the same bus. If a `fileName` is specified, the data read and written through DiskDeviceTransfer will be stored in a file by that name. The file acts as a surrogate disk, and will thus appear to be very large; however, it will be full of "holes" and thus occupy little more real disk space than the data you write to it. If the `fileName` is `NULL`, then all data written is simply thrown away, and garbage will be returned when reading (which is fine when you are running data-independent tests).

`DiskDeviceDone(int disk)` This routine is called once for each disk, when the disk is no longer needed. It will wait for any pending writes to complete, and then free up its internal data structures.

`DiskDeviceTransfer(disk, sector, nsectors, write, buffer)`

Do a disk operation, given the disk number, logical sector number on that disk (mapped to a physical sector number by the disk, taking into account sparing regions), the number of sectors to transfer, a read/write flag, and the buffer for the data. This *blocks* until the time when the transfer would be complete, that is, on return from this function logical time will have advanced appropriately.

```
DiskDeviceStart(disk, sector, nsectors, write)
DiskDeviceFinish(disk, buffer)
```

Like DiskDeviceTransfer, but split into two parts: the first part starts the disk moving to the appropriate location, and the second supplies the buffer needed to complete the transaction. The second call blocks until the transfer is complete. There should not be any other requests between a Start and a corresponding Finish.

```
DiskDeviceSync(disk)
```

This function blocks until any pending disk writes are complete (necessary when the disk has chosen to do an "immediate-reported" write).

```
DiskDeviceShape(disk, *nSectors, *sectorSize, *nTracks, *sectorsPerTrack)
```

Return the physical size of the disk.

# 3 Implementation

In this section, we describe our understanding of how the HP 97560 works, then some details about our code, and then some support mechanisms needed by the code.

## 3.1 Our understanding of the HP 97560

Our model was based on the paper by Ruemmler and Wilkes [RW94] and on the manual for the disk drive [HP91]. Although these documents do a wonderful job at describing the HP 97560, we needed to make a few assumptions beyond the information in those sources:

- When reading, the cache keeps all sectors from the beginning of the current request up through the current read-ahead point. (The alternative, throwing out sectors as they are transferred to the host, is equally plausible but from the looks of the trace data is not how the disk actually worked). The actual policy is not known; presumably for requests larger than the cache size there is some way to discard data from the cache to allow the transfer to continue.

- Read and write fences are 64 KB.

- When transferring across a sector boundary that is also a track and cylinder boundary, we assume that the track-switch cost subsumes the head-switch cost. That is, we take the maximum, rather than the sum. Apparently this is what Ruemmler and Wilkes did.
- When crossing a cylinder boundary, cylinder skew subsumes the track skew (thus, the cylinder skew is the total skew). Same reasoning as in the previous note.
- When transferring across a track or cylinder boundary, the switch time is determined to be identical to the corresponding "skew time" (skew distance divided by the rotational speed).
- Bus speed is 10 MB/s.
- It takes 50 $\mu$sec to grab the bus. We guessed this parameter.
- Disk requests on the bus are 10 bytes, derived from [HP91]. John Wilkes thinks they used 50 bytes.
- "Done messages" on the bus are 1 bytes.
- Controller overhead (for both read and write) is 2.2 msec, after [RW94], though longer than that in [HP91].
- We assume that read-ahead can be aborted in the middle of a sector transfer, which is documented in [HP91].
- The 97560 does not have segmented cache, command queueing, or multiple zones.
- We ignore thermal recalibrations.

### 3.2 Structure of the code

The main part of the code is in the DiskDevice module, spread over four files:

**diskdevice.h:** the interface to the module

**diskdevices.h:** included only by the diskdevice*.c files, this defines parameters of the disk drives

**diskdevice-model.c:** the bulk of the code for the module

**diskdevice-trivial.c:** a trivial disk model, with the same interface as the real model, and assumes a constant disk-access time. This is useful when debugging users of the disk device code, because of its speed and simplicity.

Our model is event driven. A logical clock is maintained by our support software (see the next section). When a disk request arrives (through the interface described in Section 2), an event is scheduled at the current logical time. When the event is scheduled, it grabs the bus and sends the request to the disk controller. Then the interface function sleeps (in a thread-based system, this sleep would suspend the current thread; in our one-disk stand-alone version, sleeping just calls the scheduler to process events until a "wakeup" flag is set), while each event is scheduled, events schedule more events, and the logical clock advances. Eventually, one of the events is responsible for waking up the sleeping thread, which then returns from the interface function to the user. See Figure 1 for a pictorial representation of the events and their interrelationship.

A normal disk transfer request works as follows. The user calls DiskDeviceTransfer (abbreviated DD Transfer on the chart). After the parameters are recorded in the per-disk data structure, SendCommand is called. SendCommand attempts to grab the bus, by checking the integer that indicates the owner (disk ID) of this bus. If the grab succeeds (the bus was already ours or was free), SendCommand schedules an EndCommand event at a future time based on the current time, the time needed to grab the bus, and the time needed for the command to traverse the bus. If it could not grab the bus, it enqueues a SendCommand request on the bus-wait queue (there is one for each bus). When other disks finish using the bus, they will hand the bus to this disk and reschedule the SendCommand event. The SendCommand event will of course succeed to grab the bus, and continue. This model for using the bus is repeated in other situations.

When EndCommand is eventually scheduled, it passes the bus to waiting disks, if necessary, and then calls the Controller.

The Controller, in its simplest form, works as follows (immediate-reported writes add some complexity, which we describe further below). If the new request is a **read**, it looks at the cache to determine whether there is a hit. If not, it flushes the cache, cancels any active prefetch operation, and schedules a DiskMove event to move the disk head to the appropriate location. If there was a cache hit, it flushes the cache up to the first requested sector, and schedules a ConsiderBusXfer event to get the bus transfer started. If the new request is a **write**, it cancels any active prefetch (from a preceding read operation), schedules a DiskMove event to move the head to the appropriate location, and schedules a ConsiderBusXfer event to get the bus transfer started.

There are then essentially two event loops, one for disk transfers (from cache to disk, or from disk to cache), and one for bus transfers (from cache to bus, or from bus to cache).

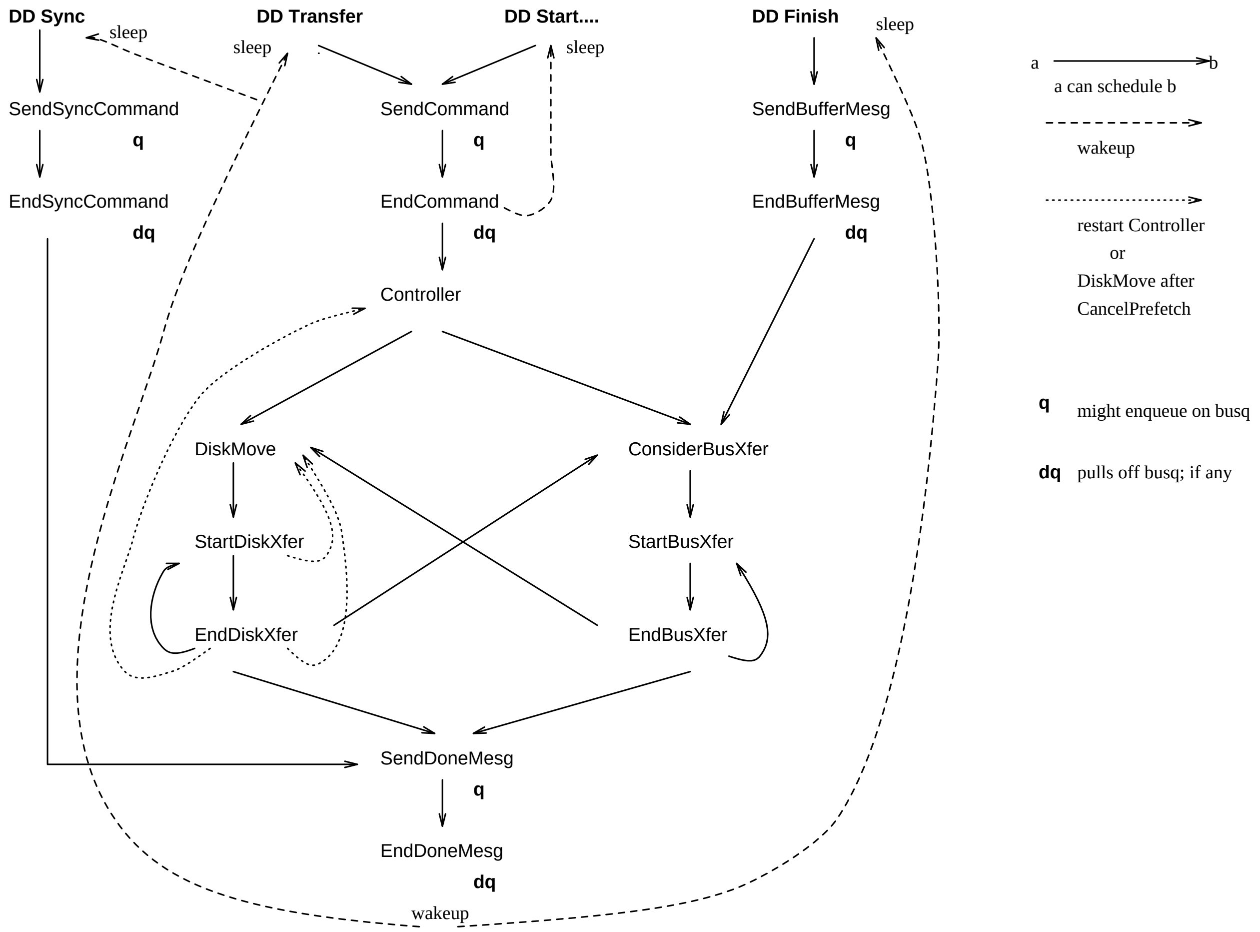


Figure 1: Event graph of our simulation model. Bold names at the top denote the entry points that may be called by the driver program. Other names represent event types, each of which is implemented as a function. Solid arrows indicate which functions can schedule which events. Dashed arrows represent waking up the thread sleeping in the entry function (in our simple environment, there are no threads, and this just returns to the caller). Dotted arrows are a relatively rare form of event scheduling. **q** annotates events that may enqueue a request on the bus queue; **dq** annotates events which may hand off the bus to an event waiting on the bus queue.

In the disk-transfer loop, DiskMove computes the time needed to move the head to the beginning of the first requested sector. It schedules a StartDiskXfer event at that time. StartDiskXfer updates the state of the cache, and schedules an EndDiskXfer at the time when that sector's transfer will be finished. EndDiskXfer again updates the state of the cache, and, if the transfer should continue, schedules a StartDiskXfer. If the transfer is complete, and is a write, it schedules a SendDoneMesg to tell the host that the transfer completed. Finally EndDiskXfer calls ConsiderBusXfer to see what can be done (either to start transferring a newly read sector, or to refill the cache after having written a sector to disk).

The bus-transfer loop is similar, in that it transfers one sector at a time in a loop between StartBusXfer and EndBusXfer. ConsiderBusXfer (sometimes, sometimes called directly) decides whether a bus-transfer loop should be started; if so, it grabs the bus and schedules StartBusXfer. Starting a bus transfer depends on a complicated set of circumstances: we are not already using the bus, the controller has indicated it wants to use the bus (not true, for example, when prefetching), the host has a buffer to hold the data (not always true in the split-phase interface we provide), and, if reading, either the last sector is ready or the cache has filled with plenty of sectors (the "read fence"), or if writing, the last sector has not been transferred and the cache is sufficiently empty (the "write fence").

EndBusXfer either schedules another StartBusXfer, continuing the loop, or (if a read request) schedules SendDoneMesg to indicate that the transfer is complete. EndBusXfer may schedule DiskMove; this happens when the disk-transfer loop had to stop because it was reading and filled the cache (which is drained by the bus-transfer loop) or was writing and emptied the cache (which is filled by the bus-transfer loop). DiskMove restarts the loop after the disk has rotated into the correct position.

SendDoneMesg grabs the bus and sends a short "finished" signal to the host, by scheduling EndDoneMesg. EndDoneMesg wakes up the sleeping caller.

There are a few rare transitions. When a prefetch request is canceled, sometimes the request can be located and pulled from the event queue, but sometimes the event cannot be located and is eventually scheduled. A flag causes the event (usually StartDiskXfer) to abort and call DiskMove, which moves the disk to the new location.

Immediate-reported writes are more complex. Immediate reporting means that done message is sent to the host as soon as the last sector has been transferred (by the bus) into the cache, rather than on to disk. Furthermore, when a sequentially contiguous write request arrives soon after an immediate-reported write, the new data is appended to the cache, and will be transferred without any rotational delay between the two requests. If a read or non-contiguous write arrives after an immediate-reported write, it must wait for the preceding write to complete before beginning any bus or disk transfer of its own. Our interface assumes that all writes may be immediate-reported, and provides a separate DiskDeviceSync command that simply waits for any outstanding I/O. This interface is consistent with the SCSI interface [HP91]. If a synchronous write is desired, the pair DiskDeviceTransfer and DiskDeviceSync accomplish the same thing with no additional overhead.

The Controller becomes more complex when immediate-reported writes are considered. First, if there is a outstanding disk-transfer loop in progress, and the new request is a read or non-contiguous write, the new request must wait. To accomplish this wait, the Controller sets a flag (`restartController`) and quits. In the case of a contiguous write, where it simply adjusts the parameters and considers starting a bus transfer to fetch the new data. Otherwise it works as before.

The Controller will be restarted (i.e., the Controller event rescheduled) by EndDiskXfer when it detects that the disk-transfer loop is complete and the `restartController` flag is true. The Controller starts the new request as before.

Split-phase requests (DiskDeviceStart/DiskDeviceFinish) are also a little bit more complicated. EndCommand wakes up the sleeping DiskDeviceStart, so that the caller can continue as soon as the command arrives at the controller. Later, the caller is expected to provide a buffer by calling DiskDeviceFinish. This sends a "buffer message", essentially a signal to the controller that the bus transfer can begin. StartBufferMesg simply schedules EndBufferMesg at the appropriate time, and EndBufferMesg sets a flag in the controller to indicate that the bus transfer can begin and then calls ConsiderBusXfer. EndDoneMesg wakes up the DiskDeviceFinish as usual.

**Invariants.** Each disk has five key variables in its state. These keep track of the bus- and disk-transfer loops, and the contents of the cache. They are

**FCS:** first cached sector

**CBX:** current bus transfer

**NBX:** next bus transfer

**CDX:** current disk transfer

**NDX:** next disk transfer

Essentially, NBX is the next sector to transfer on the bus and NDX is the next sector to transfer on the disk. If NDX $>$ CDX, then there is a sector (CDX) actively being transferred on the disk; same for the bus. The disk-transfer leads the bus-transfer when reading, and the bus-transfer leads the disk-transfer when writing. Because the cache is a FIFO buffer between the disk and the bus, they can never get too far apart. Indeed, the cache begins with sector FCS, which may (in large requests) be quite far behind the bus and disk activity. See Figure 2 for a pictorial representation of some of these invariants, which are described in detail below:

- When reading,
  - FCS $\leq$ CBX $\leq$ NBX $\leq$ CDX $\leq$ NDX
  - FCS is not actually in the cache unless FCS $<$ NBX
  - (CDX-FCS+1) $\leq$ cache size
- When writing,
  - FCS $\leq$ CDX $\leq$ NDX $\leq$ CBX $\leq$ NBX
  - FCS is not actually in the cache unless FCS $<$ NDX
  - (CBX-FCS+1) $\leq$ cache size
- NBX$-$CBX = 0 or 1
  - if 0, no bus transfer is active
  - if 1, sector CBX is going across the bus
- NDX$-$CDX = 0 or 1
  - if 0, no disk transfer is active
  - if 1, sector CDX is transferring to/from the disk

### 3.3 Discrete-event support

In addition to the code for the disk model itself, we have a few modules to support the discrete-event simulation. By default, these are very simple mechanisms to support "stand-alone" use of the disk model. They can be replaced, however, by interface modules that can integrate the disk model into another simulation environment. These support modules include

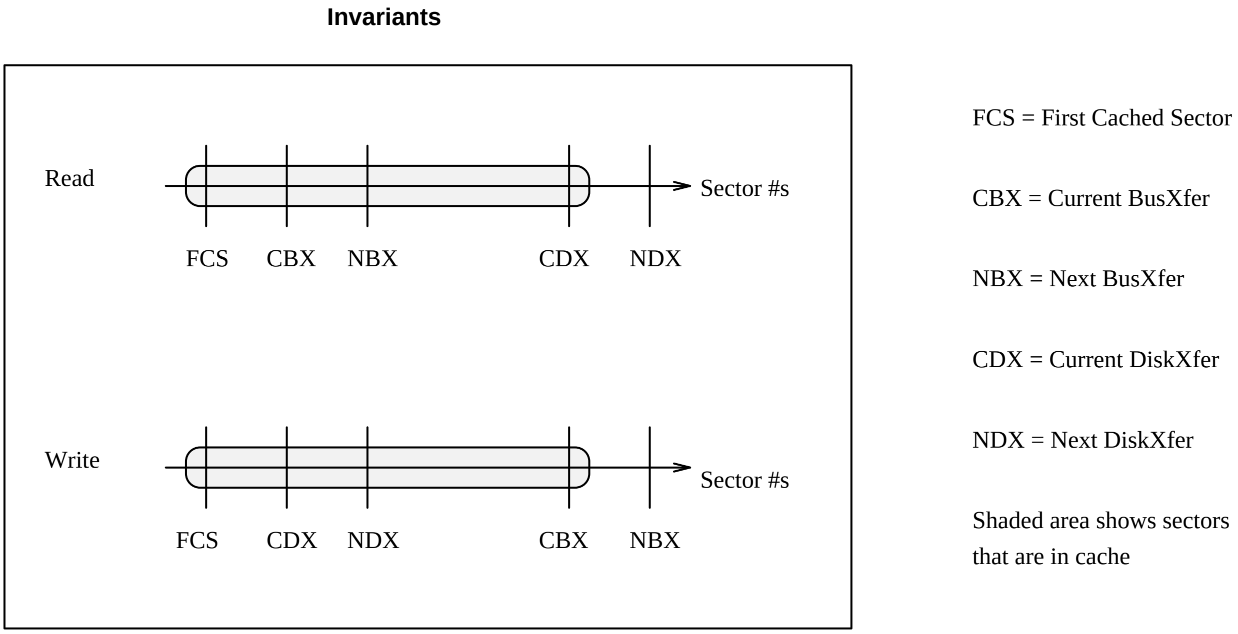


Figure 2: Invariants maintained by our disk model. Five key variables keep track of the state of this disk, in particular, the state of the disk transfer (current and next sector, CDX and NDX), the state of the bus transfer (current and next sector, CBX and NBX), and the state of the cache (first cached sector (FCS), and the others). Details of invariants are given in the text.

**diskevent.c:** Functions to schedule a given disk event at a given time, to cancel an event that was previously scheduled, and to sleep and wake up the caller's thread.

**queue.c:** a simple queue used for tracking those disks that wish to use the bus.

**heap.c:** a priority queue (heap) used to schedule events chronologically.

**modularize.c:** a few miscellaneous functions

We have also integrated the disk model into the Proteus parallel-architecture simulator [BDCW91], where we simulate a parallel file system involving many disks and buses [Kot94].

# 4 Validation

To validate our model, we used a trace-driven simulation, choosing trace data from the same set of disk traces used by Ruemmler and Wilkes in their study.[1] In particular, we chose the "snake" traces from 4/25/92 through 4/30/92, disk numbers 5 and 6. After filtering the trace files for events from those disk drives, and sorting them by the time the (real) disk started the request, we ran

[1] Kindly provided to us by John Wilkes and HP. Contact John Wilkes at wilkes@hplabs.hp.com for information about obtaining the traces.

them through our disk model. In the process we assumed that all writes that were marked both "synchronous" and "metadata" should call DiskDeviceSync after DiskDeviceTransfer, to force the caller to wait until the write was completely on disk before issuing the next request.[2] Otherwise, the writes were "immediate," i.e., the caller could continue while the disk write executed in the background. Between requests, we waited the same amount of time as was recorded in the trace from the end of the previous request until the start of the current request.[3]

We discovered that the real disk produced some extremely long access times (some nearly 400 msec), which our model never approached. Our access-time distribution curve was visibly close, but the demerit figure was poor. The curves in [RW94] end at 50 msec. After consulting with Chris Ruemmler, we discovered that he had filtered the traces to exclude these "long" requests, assuming that they were due to thermal recalibration, something they (and we) did not model. Indeed, these events are rare ($\ll 1\%$ of all events). After filtering the traces to discard all requests in which the real disk took $> 100$ msec, then re-running our simulation, we obtained a very close match.

We simulated each disk on each day independently, combining the sets of modeled access times into one big set of nearly 665,000 accesses. From the trace we also had the set of real access times on the same requests. Our mean access time was 12.434 msec, with a demerit figure of 0.484, and a demerit percentage of 3.89% (see [RW94] for a definition of these measures). Figure 3 shows the distributions; Figure 4 shows the same distributions on a 0 to 50 msec scale, for comparison with [RW94].

We also used a set of microbenchmarks, based on regular access patterns (reading and writing sequentially, reading increasingly large blocks but always starting from sector 0, *etc.*). Plots of these access times (not shown) allowed us to "check" our model against the intuition.

## Availability

The software is available for ftp at cs.dartmouth.edu in `pub/diskmodel`, or on the WWW at `http://www.cs.dartmouth.edu/cs_archive/diskmodel.html`.

The software is written in ANSI C. It does not use threads, but a threads package (and some minor porting) would be needed to use more than one disk in parallel (there is no asynchronous interface, so you would need a thread for each disk so that blocking on one disk access does not

[2] The trace format was not completely clear on the semantics here, but this made sense and, from looking at the request times, seemed to agree with how the real disk behaved.

[3] There are other plausible alternatives. For example, we could have waited until the clock reached the same time as the start of the current request in the trace.

prevent you from using other disks).

## Acknowledgements

Song Bac Toh, an undergraduate, wrote most of the code for the disk model, and Sriram Radhakrishnan, another undergraduate, wrote most of the code for trace-driven validation of the model. Many thanks to John Wilkes, Chris Ruemmler, and HP for the wonderful paper [RW94], trace data, and answers to our many questions.

## References

[BDCW91] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. Proteus: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR–516, MIT, September 1991.

[HP91] Hewlett Packard. *HP97556/58/60 5.25-inch SCSI Disk Drives Technical Reference Manual*, second edition, June 1991. HP Part number 5960-0115.

[Kot94] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Operating Systems Design and Implementation*, June 1994. Submitted extended abstract.

[RW94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, March 1994.

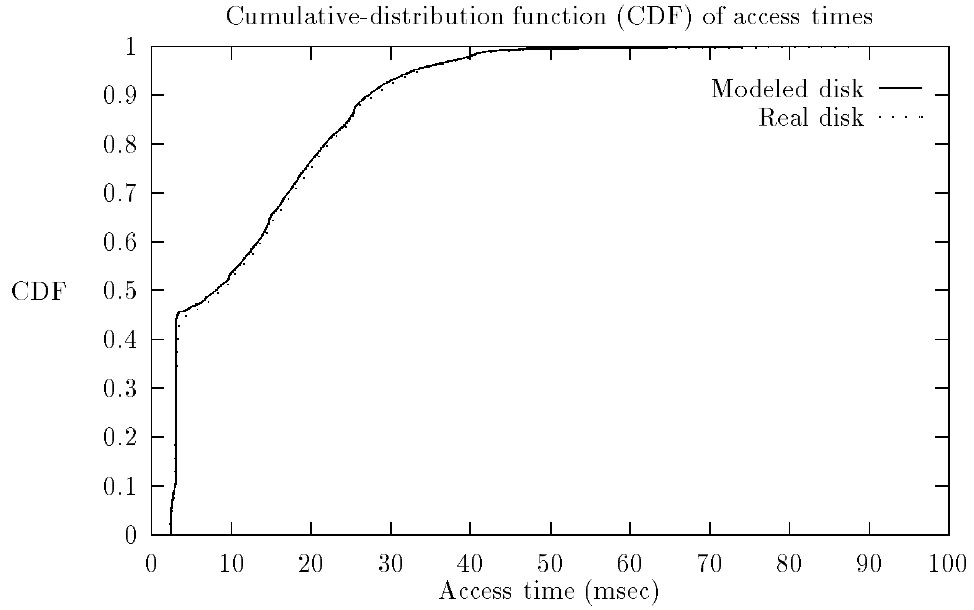


Figure 3: Distribution of all access times from six days and two disks, for both our model and the original disks. There were 664,995 accesses. The overall demerit percentage was 3.89%.

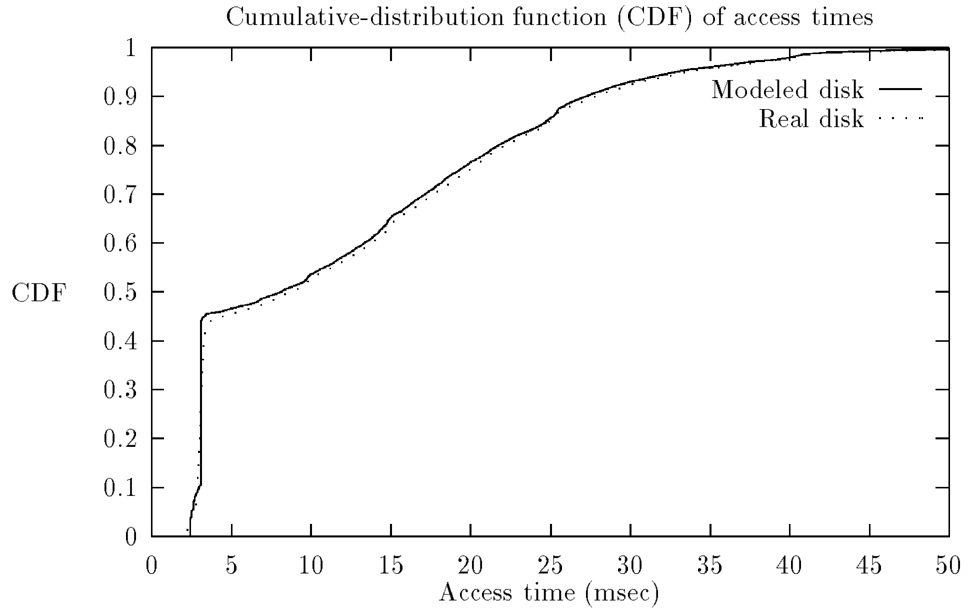


Figure 4: Same as the previous figure, but limiting the $x$-axis to 50 msec.