# Timing-accurate Storage Emulation

John Linwood Griffin, Jiri Schindler,
Steven W. Schlosser, Gregory R. Ganger

July 2001
CMU-CS-01-146

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

## Abstract

*Timing-accurate storage emulation fills an important hole in the set of common performance evaluation techniques for proposed storage designs: it allows a researcher to experiment with not-yet-existing storage components in the context of real systems executing real applications. As its name suggests, a timing-accurate storage emulator appears to the system to be a real storage component with service times matching a simulation model of that component. This paper promotes timing-accurate storage emulation by describing its unique features, demonstrating its feasibility, and illustrating its value. A prototype, called the Memulator, is described and shown to produce service times within 2% of those computed by its component simulator for over 99% of requests. Two sets of measurements enabled by the Memulator illustrate its power: (1) application performance on a modern Linux system equipped with a MEMS-based storage device (no such device exists at this time), and (2) application performance on a modern Linux system equipped with a disk whose firmware has been modified (we have no access to firmware source code).*

We thank the members and companies of the Parallel Data Consortium (including EMC, Hewlett-Packard, Hitachi, Intel, LSI Logic, Lucent, Network Appliance, PANASAS, Platys Communications, Seagate, Snap Appliances, Sun Microsystems, and Veritas) for their interest, insights, and support. We also thank IBM for supporting our research efforts.

**Keywords:** Disk scheduling, storage systems

## 1 Introduction

Despite decades of practice, performance evaluation of proposed storage subsystems is almost always incomplete and disconnected from reality. In particular, future storage technologies and potential firmware extensions usually cannot be prototyped by researchers, and so any evaluation must rely upon simulation or analytic models of the prospective subsystem. Unfortunately, this reliance commonly limits consideration of real application workloads and complex “real system” effects, both of which can hide or undo benefits predicted by simulating storage components in isolation. For this reason, such localized evaluation has long been considered unacceptable in other disciplines, such as networking, architecture, and even file systems.

Timing-accurate storage emulation offers a solution to this dilemma, allowing simulated storage components to be plugged into real systems, which can then be used for complete, application-based experiments. As illustrated in Figure 1, a *storage emulator* transparently fills the role of a real storage component (e.g., a SCSI disk), correctly mimicking the interface and retaining stored data to respond to future reads. A *timing-accurate* storage emulator responds to each request after its simulator-computed service time passes; the performance observed by the system should match the simulation model. To accomplish this, the emulator must synchronize the simulator’s internal time with the real-world clock, inserting requests into the simulator when they arrive and reporting completions when the simulator says they are done. If the simulator’s model represents a real component, the system-observed performance will be of that component. Thus, the results from application benchmarking will represent the end-to-end performance effect of using that component in the real system.

This paper makes a case for timing-accurate storage emulation and demonstrates that it works in practice. It describes general design issues and details the implementation of our prototype emulator in Linux. Our original goal was thorough evaluation of operating system algorithms for not-yet-existing MEMS-based storage devices [10, 11]—this led to the prototype’s name: *Memulator*. The Memulator integrates the DiskSim simulator [9], a real-time timing loop, and a large RAM cache to achieve flexible, timing-accurate storage emulation. It can emulate any storage component that DiskSim can simulate, including

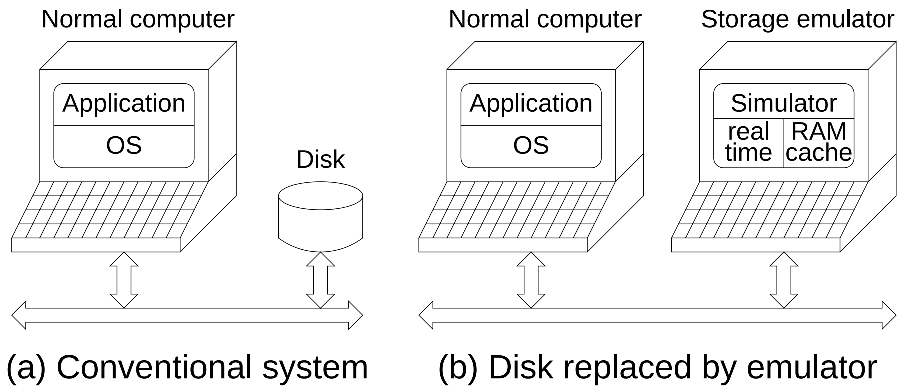


Figure 1: **A system with (a) real storage or (b) emulated storage.** *The emulator transparently replaces storage devices in a real system. By reporting request completions at the correct times, the performance of different devices can be mimicked, enabling full system-level evaluations of proposed storage subsystem modifications.*

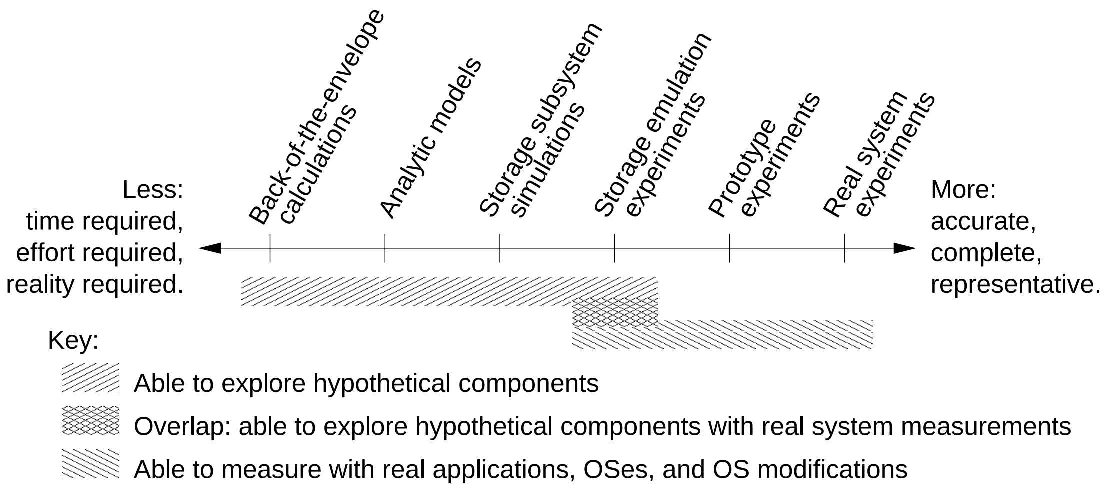

Back-of-the-envelope calculations
Analytic models
Storage subsystem simulations
Storage emulation experiments
Prototype experiments
Real system experiments
Less: time required, effort required, reality required.
More: accurate, complete, representative.
Key:
Able to explore hypothetical components
Overlap: able to explore hypothetical components with real system measurements
Able to measure with real applications, OSes, and OS modifications

Figure 2: **Storage performance evaluation techniques.** *This illustration linearizes techniques in a spectrum from the* ⟨*quickest,easiest,most flexible*⟩ *to the most* ⟨*accurate,complete,representative*⟩*. In this spectrum, storage emulation provides the unique ability to explore nonexistent storage components in the context of full systems executing real applications.*

MEMS-based storage, disk arrays, and many modern disk drives. Calibration measurements indicate that the Memulator's response times are within 2% of the DiskSim times for over 99% of requests. Using DiskSim's validated disk models, we also verify that system performance is the same with the Memulator or a real storage device.

We illustrate the power of timing-accurate storage emulation with two experiments that the Memulator makes possible. First, we measure how MEMS-based storage would affect application performance on a current Linux system; since fully-functioning MEMS-based storage devices are still years away, this experiment is only possible with emulation. Second, we measure how an extension (zero-latency reads) to disk firmware would affect application performance on a Linux system; since we have no access to firmware source code, we can only do this with emulation. We also discuss a third type of experiment, interface extensions, that requires changes to both the host OS and the storage subsystem; without emulation (or complete implementation), thorough evaluation of interface extensions is not possible.

The remainder of this paper is organized as follows. Section 2 makes a case for timing-accurate storage emulation. Section 3 discusses the design of timing-accurate storage emulators in general. Section 4 describes the Memulator in detail. Section 5 validates the response times of the Memulator relative to simulated device performance. Section 6 describes experiments enabled by the Memulator. Section 7 summarizes this paper's contributions.

## 2 A case for emulation

Storage emulation is rarely used for performance evaluation of prospective storage system designs. This section makes a case for more frequent use, arguing that timing-accurate storage emulation offers a unique performance evaluation capability: experimentation with as-yet-unavailable storage components in the context of real systems. Such experimentation is important because complex system characteristics can hide or reduce predicted benefits of new storage components [7]. Further, some new storage architectures and interfaces require both OS modifications and new (or modified) storage components—until the new components are available, only emulation allows such collaborative advances to be tested and their performance evaluated.

## 2.1 Storage performance evaluation

Figure 2 illustrates a spectrum of techniques for evaluating storage designs, ranging from quick-and-dirty estimates to real application measurements on a complete system. Techniques to the left generally demand less of the evaluator: less effort to set up and employ, less time to produce a result, and less need for the evaluated storage system to be feasible. Techniques to the right generally produce more believable results: more accurate, more inclusive of complex system effects, and more representative of the effects under real workloads.

The six techniques shown are each appropriate in some circumstances, as each offers a different mixture of these features. For example, storage simulation allows hypothetical storage systems to be evaluated quickly and efficiently. Even futuristic technologies and modifications to proprietary firmware can be explored. Simulation results, however, must be taken with a grain of salt, since the simulation may abstract away important characteristics of the storage components, overall system, or workload. In particular, representative workloads are rarely used, since synthetic generation is still an open problem [6], I/O traces ignore system feedback effects [7], and available traces are often out-of-date—in fact, many storage researchers still rely on the well-known "HP traces" from 1992 [20]. As a different example, experimenting with prototypes allows one to evaluate designs in the context of full systems and real workloads. Doing so, of course, requires considerable investment in prototype development and experiment configuration.

As indicated in Figure 2, storage emulation offers an interesting mix of features: the flexibility of simulation and the reality of experimental measurements. That is, storage emulation allows futuristic storage designs to be evaluated in the context of real OSes and applications. This enables two types of experiments. First, end-to-end measurements can be made of the effects of non-existent storage components in existing systems. Such components are usually simulated in isolation and evaluated under non-representative workloads. Second, end-to-end measurements can be made of the effects of non-existent storage components in *modified* systems. For example, storage interface changes often require that both the storage components and the OS be modified to utilize the new interface. Experimentation is impossible without the ability to modify both components, which is a very real problem with the proprietary firmware of most disks and disk array controllers. Section 6 explores concrete examples of both types of experiments.

We are aware of only one other technique offering a similar mix of features: complete machine simulation [18]. In this technique, the hardware of a computer system is simulated in enough detail to boot a real OS and run applications. If the simulation progresses according to timing-accurate models of the key system components (e.g., CPUs, caches, buses, memory system, I/O interconnects, I/O components), it can be used for performance evaluation. Because it boots a real OS and runs real applications, a complete machine simulator enables the same types of experiments as storage emulation. Further, by manipulating simulator parameters, the effects of new storage devices on hypothetical machines (e.g., with 10GHz CPUs) can be evaluated [19, 21]. Unfortunately, substantial effort is required to build and maintain a complete machine simulator, both in terms of correctly executing programs and correctly accounting for time. For example, the SimOS machine simulator required extensive effort to create and validate; just a few years later, its hardware models are out of date, the CPU instruction set it emulates is being phased out, and source code for the OS that it boots is difficult to acquire. In addition, these simulators usually run more slowly than real systems, increasing evaluation time. Storage emulation does not share these difficulties.

## 2.2 Related emulation

In a sense, storage emulation is commonplace. For example, the standard SCSI interface allowed disk arrays to rapidly enter the storage market by supporting a disk-like interface to systems. Similarly, the NFS remote procedure call (RPC) interface allowed dedicated filer appliances [12] to look like traditional NFS file servers. In addition, we have been told anecdotal stories of emulation's use in industry for development and correctness testing of new product designs. However, these examples represent only the "storage emulation" half of timing-accurate storage emulation.

The "timing-accurate" half has been much utilized by networking researchers [1, 5, 17]. Timing-accurate network emulation parallels our description of timing-accurate storage emulation: real hosts interconnected by the emulated network observe normal packet send/receive semantics and performance that accurately reflects a simulation model. The observable performance effects include propagation delays, bandwidths, and packet losses. Like timing-accurate storage evaluation, timing-accurate network emulation enables real system benchmarking that would not otherwise be possible—in particular, deploying a substantial network just for experiments is simply not feasible.

We are aware of only a few previous cases of timing-accurate storage emulation being used for performance evaluation. The most relevant example is Wang et al.'s evaluation of eager writing [23]. Under eager writing, data is written to a disk location that is close to the disk head's current location. To evaluate the benefits of having disk firmware support for eager writing, Wang et al. embedded a disk simulator in Solaris 2.6, augmented it with a RAMdisk, and arranged (by using `sleep`) to have completions reported after delays computed by the simulator. Although some details differ, this is similar to the Memulator's design. A less direct example is the common practice of emulating non-volatile RAM by simply pretending that normal RAM is non-volatile [4, 8]. Although this would be unacceptable in a production system, such pretending is fine for performance experiments.

A central purpose of this paper is to promote timing-accurate storage emulation as a first-class tool in the storage research toolbox. Towards this end, we describe its unique capabilities, demonstrate its relatively straightforward realization, and illustrate its power with several experiments that we could not otherwise perform.

## 3 Emulator Design

A timing-accurate storage emulator must appear to its host system to be the storage subsystem that it emulates. Doing so involves three main tasks. First, the emulator must correctly support the protocols of the interface behind which it is implemented. Second, the emulator must complete requests in the amount of time computed by a model of the storage subsystem. Third, the emulator must retain copies of written data to satisfy read requests. This section describes how these three tasks are handled and the steps an emulator goes through to service storage requests.

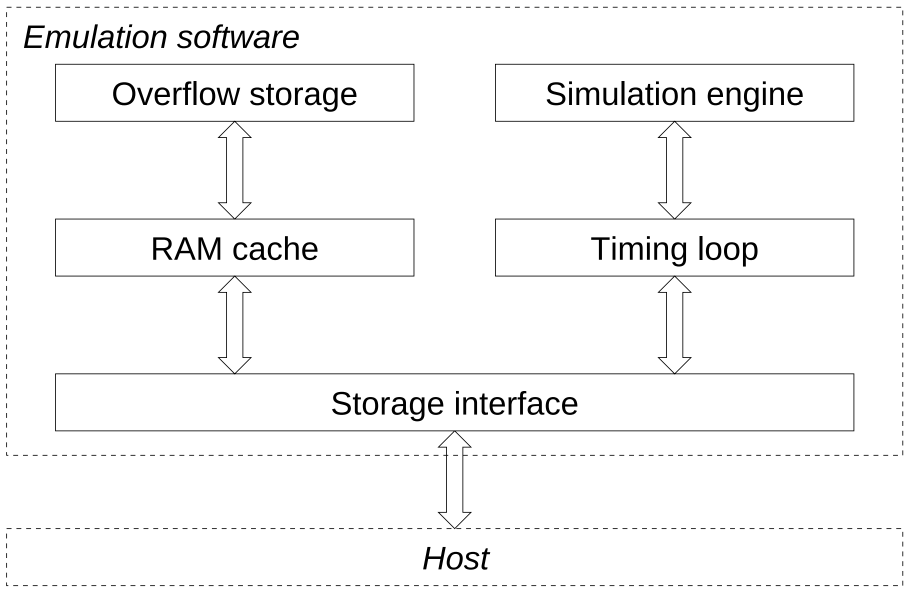


Figure 3: **Emulation software internals.** *The five components inside the "storage emulation software" box comprise the three primary emulator tasks: communications management (the storage interface), timing management (the simulation engine and timing loop), and data management (the RAM cache and overflow storage).*

## 3.1 Emulator components

Figure 3 shows the internals of a timing-accurate storage emulator. This section describes how the components of the emulator work to satisfy the three tasks: communications management (the storage interface), timing management (the simulation engine and timing loop), and data management (the RAM cache and overflow storage).

### 3.1.1 Communications management

The storage interface component connects the emulator to the host system. As such, it must export the proper interface. The storage interface ensures that requests are transferred to and from the host according to the emulated protocol. Incoming requests are parsed and passed to the other emulator components, and outgoing messages are properly formatted for return to the host. In addition to servicing requests, the storage interface must respond appropriately to exceptional cases such as malformed requests or device errors.

In response to a read or write request, the storage interface parses the request, checks its validity, and then passes it to the timing and data management components of the emulator.

In some cases, it may have to interact further with the host (e.g., for bus arbitration or if the emulated device supports disconnection). In addition to reads and writes, the emulator must also support control requests that return information about the emulated drive such as its capacity, status, error condition, etc. In practice, a subset of often-used control commands usually suffices. When a request is completed, the response is formatted appropriately for the emulated protocol and forwarded to the host through the storage interface.

### 3.1.2 Timing management

The simulation engine and timing loop work together to provide the timing-accurate nature of the emulation. Specifically, the simulator determines how long each request should take to complete, and the timing loop ensures that completion is reported after the determined amount of time.

There are two ways that the simulation engine and timing loop can interact. One approach keeps the two separate: when a request arrives, the timing loop calls the simulator code once to get the service time. In this approach, the simulator code takes the real-world arrival time and the request details, and it returns the computed service time. After the appropriate real-time delay, the timing loop tells the storage interface component to report completion. Wang et al.'s emulator-based evaluation of eager writing used Kotz's disk simulator [15] in this way.

Although it is straightforward, this first approach often does not properly handle concurrent requests. For example, a new request arrival may affect the service time of outstanding requests due to bus contention, request overlapping, or request scheduling. A more general approach is to synchronize the advancement of the simulator's internal clock with the real-world clock. This synchronization can most easily be done with event-based simulation.

An event-based simulator breaks each request into a series of abstract and physical events: request arrival, controller think time complete, disk seek complete, read of sector $N$ complete, and so on. Each event is associated with a time, and an event "occurs" when the simulator's clock reaches the corresponding time. Event occurrences are processed by simulation code that updates state and schedules subsequent events. For example, the "controller think time complete" event may be scheduled to occur a constant time after the "request arrival" event.

To synchronize an event-based simulation with the real world, the emulator lets the timing loop control the simulator clock advancement. When each event completes, the simulator engine notifies the timing loop of the next scheduled event time. The timing loop waits until that time arrives, then calls back into the simulator to begin processing the next event. If a new request arrives, a "request arrival" event is prepended to the simulator's event list with the current real-time, and the timing loop calls into the simulator immediately. When a "request complete" event ultimately occurs, the simulator engine notifies the storage interface.

In practice, the request arrival and completion times must be skewed slightly to account for processing and communication delays. The arrival time of a request is adjusted backwards slightly to account for the delay in receiving the request. Likewise, the simulator runs slightly ahead of the real-world clock so that the storage interface will start sending completion messages early enough for them to arrive on time. Clearly, an additional requirement is that the simulation computations themselves be fast enough that they do not delay completion messages; the computation time for any given request must be lower than the computed service time.

### 3.1.3 Data management

In addition to providing accurate timing of requests, emulation software must provide a consistent view of stored data. This is satisfied by the combination of a RAM-based block (sector) cache and overflow storage for paging blocks from the cache. These components act as a conventional memory manager: groups of blocks can be grouped into "pages" that are evicted from or promoted into the cache. The overflow storage is only necessary for workloads requiring active storage in excess of the memory allocated to the emulation software. Possible implementations of the overflow storage include paging to one or more locally-attached disk drives, or paging to shared network-based RAM [2].

Data transfers from overflow storage may not complete quickly enough when emulating a high-performance device. When this is the case, cache preloading schemes may be necessary to ensure high RAM cache hit rates. These schemes can take advantage of the repeatability of experiments. For example, a workload could be initially run solely to generate a trace of

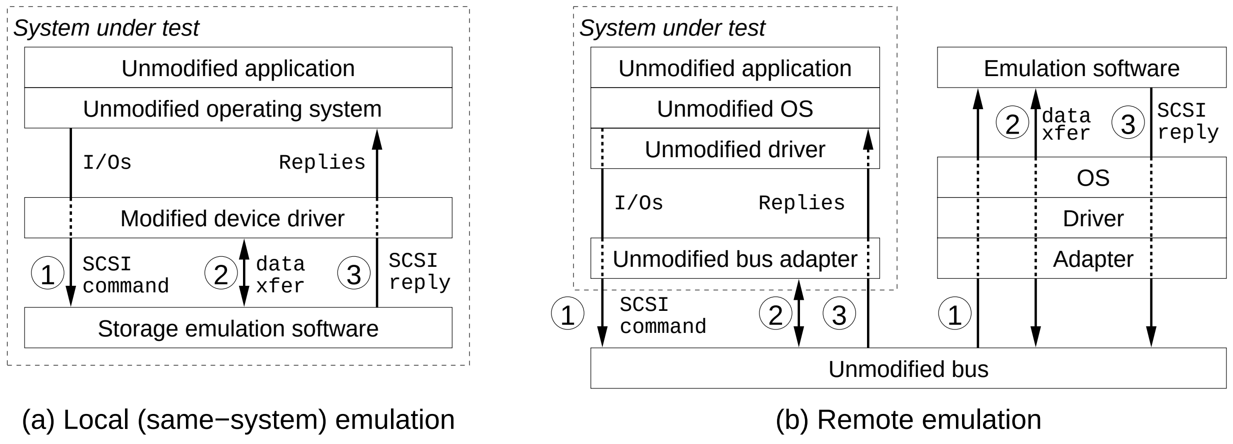


Figure 4: **Communication paths when emulation is run (a) locally or (b) remotely.** *When run locally, emulation software communicates directly with a modified device driver in the kernel. Under remote emulation, all modifications take place outside the system under test, eliminating the impact of the emulation overheads.*

accessed blocks, then run a second time using that trace to intelligently preload the cache throughout execution.

Since a timing-accurate storage emulator is used only as a performance evaluation tool and not as a production data store, some persistence characteristics can be relaxed to increase performance. For example, write-back caching can be used to avoid costly overflow storage delays. If the system crashes and data is lost, the experiment can be re-run.

## 3.2 Host system interactions

Figure 4 shows the two most natural points at which to integrate a storage emulator into a host system. In the first, the device driver is modified to communicate directly with emulation software rather than with real storage components. Although this does involve some modifications to the host system, they are restricted to the device driver. In the second, the host system is left unmodified, and the emulation software runs on a second computer attached to the host via a storage interconnect. The second computer responds just like a real storage device would. Both integration points leave intact the application and OS software which is doing the real work and generating storage requests. Both also share a simple 3-step interface between the storage emulator and the rest of the system.

**Step 1: Send request to the emulator.** When a read or write request arrives at the device driver, it is directed to the emulated device. In the case of local emulation, the device driver is modified to be aware of the emulation software and explicitly delivers the request to it. A device that is emulated remotely does not need a modified device driver; requests are sent unmodified across the bus to the emulation machine which in turn delivers the request to the emulation software located there. Once the emulation software (either local or remote) has the request, it issues it to the simulator engine to determine how long the request should take to complete.

**Step 2: Transfer data between the host and emulator.** The emulation software begins transferring data. In the case of a read request, data is transferred from the RAM cache to the host. In the case of a write request, data goes from the host into the RAM cache and is saved to service future reads. Data transfer should usually begin soon after the request arrives, since all data must be transferred before the completion time computed by the simulator in Step 1. A local emulator can pass pointers to data in its RAM cache directly to the modified device driver. The driver then copies data to or from the appropriate kernel buffers. A remote emulator sends data over the bus to the host.

**Step 3: Send reply to the device driver.** The emulation software waits until the request service time as determined in Step 1 elapses. At this point, all data should be transfered either from or to the host and a completion interrupt must be delivered to the OS. In the remote case, the completion message is sent over the bus, just as with a normal storage device, and the unmodified device driver deals with it appropriately. In the local case, the emulation software directly notifies the device driver that the request is complete at the device level. The driver then calls back into the operating system to complete the request at the system level.

The local design works well in practice and allows for extra communication paths between the operating system and emulator. For example, the device driver can measure perceived request service times and communicate these to the emulator, enabling the emulator to refine its model of communications overheads. In addition, this architecture enables evaluation of nonstandard device interfaces (such as freeblock requests or exposed eager writes) as discussed in Section 6.3.

However, a local emulator will have a direct impact on the system under test. Device driver modifications are necessary for communications with the emulator, and extra CPU time and memory are used to run the emulation software, which could perturb the host's workload. Using a dual-processor machine, with one CPU dedicated to emulation and with added memory dedicated to the RAM cache, will mitigate the overhead, but some interference is inevitable. A remote emulator avoids these perturbations completely by performing the emulation on separate, dedicated hardware. In this case, host overheads are eliminated and no modifications are required in the host's device driver.

In addition to device-specific delays, a local emulator must account for bus delays, since there is no physical bus between the host and the emulator. A remote emulator that is physically attached to the host via a bus need not calculate such delays, unless it is emulating a different storage interconnect.

## 4 The Memulator

This section describes the implementation of the Memulator, our timing-accurate local storage emulator for the Linux 2.4 operating system. The emulation software runs as a user-level application on the system under test and communicates with a modified SCSI device driver, as illustrated in Figure 4(a).

The modified device driver is a low-level component in the Linux SCSI subsystem, dynamically loaded as a kernel module when the Memulator is initialized. The driver accepts SCSI requests (`Scsi_Cmnd` structures) from the Linux kernel via the standard SCSI mid-to-low-level `queuecommand()` interface and passes these on to the storage interface as described below. When a request is complete, the driver notifies the kernel using the standard `scsi_done()` mid-level callback.

The Memulator's storage interface communicates with the driver via modified system calls on the special device file `/dev/memulator`. The `poll()` system call is used to notify the storage interface that a new request has arrived in the driver. `read()` is then used to transfer the 12-byte SCSI command, target, logical unit number, and a unique request identifier for that request to the storage interface. Upon receipt, the timing loop immediately

| Command | Function |
|---|---|
| READ (6 and 10) | Read data from device |
| WRITE (6 and 10) | Write data to device |
| TEST UNIT READY | Check if device online |
| INQUIRY | Get device parameters |
| READ CAPACITY | Get device size in sectors |
| REQUEST SENSE | Get details of last error |

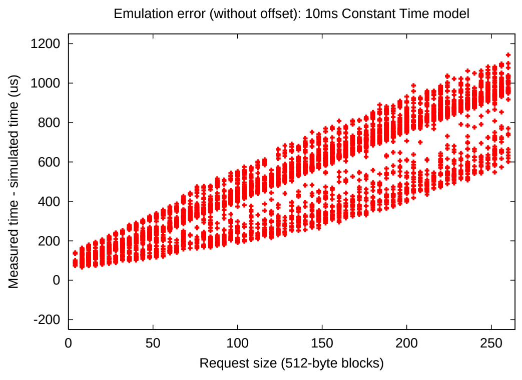
Table 1: **Required SCSI command support.** *This command set must be implemented for an emulator to interact with the Linux 2.4 kernel.*

prepends an "arrival" event for the new request to the DiskSim event queue (with an offset arrival time, described below), and the device driver immediately copies the requested data between the user and kernel memory buffers. `write()` is used to notify the device driver when the request is complete.

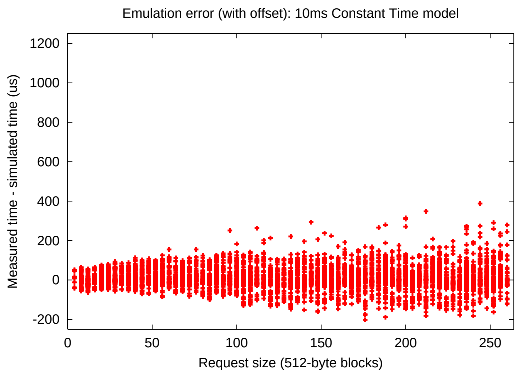
When invalid opcodes, out-of-range requests, or invalid target/LUN pairs are received during the `read()` phase, the Memulator's storage interface generates the appropriate sense code and immediately returns an error condition to the device driver via `write()`. The SCSI commands supported by our prototype are shown in Table 1. These commands are sufficient to allow Linux to mount and use Memulator devices like SCSI disks.

As discussed in Section 3.1.2, the request arrival times are skewed slightly by the timing loop to account for processing and communications overheads. Without this adjustment, request times at the storage interface will often be in error by a variable amount of time (Figure 5(a), (c), and (e)). This error is a function of both request type (read or write) and request size[1]. To compensate for the error, we determine the "arrival time offset" empirically by calculating the average difference in simulated and measured Memulator request times for different request types and sizes, and feed this information back into the Memulator (Figure 5(b), (d), and (f)).

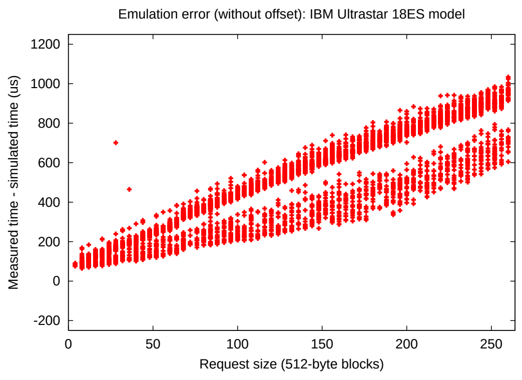
[1]It is possible that the error's dependence on request size is an artifact of extra data copies inside the Linux SCSI generic (SG) interface we use to measure the error.



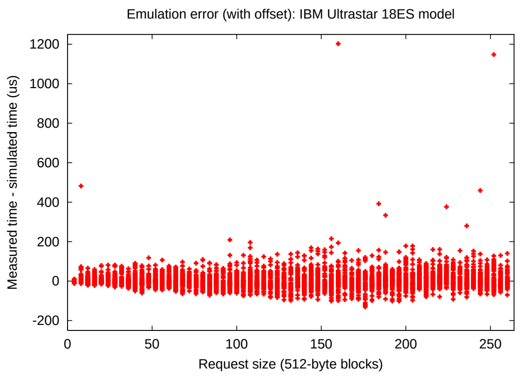
(a)



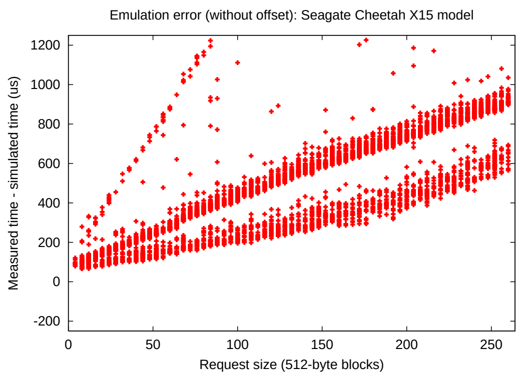
(b)



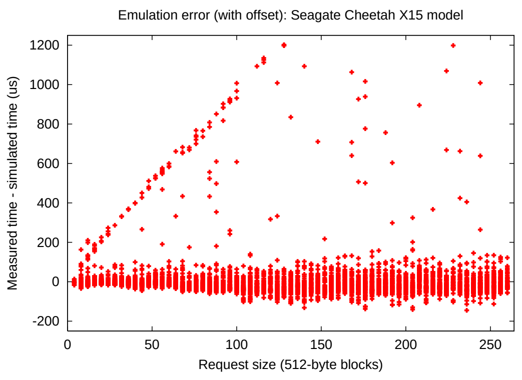
(c)



(d)



(e)



(f)

Figure 5: **Offsetting system overheads to reduce emulation error.** *Each graph shows 4,000 requests: 2,000 from the "random uniform" workload and 2,000 from the "mixed uniform" workload. The graphs on the left show the effect of OS data copies and scheduling delays. The two distinct curves are caused by different OS-level delays for read requests vs. write requests (the read curve is the upper curve), not because of the different workloads. The graphs on the right show the same workloads after compensating with the arrival time offset function. Negative values mean the request finished earlier in real time than the time specified by the simulator. The noise in graphs (e) and (f) is caused by cases where the simulation model of the Cheetah X15 takes too long to run; these can be addressed by tuning the simulator.*

## 5 Memulator Accuracy

This section presents two evaluations of Memulator accuracy. First, it shows that the Memulator accurately reflects the timings of a simulated storage device. Second, it shows that this timing-accuracy can translate into accurate emulation of a real storage device.

### 5.1 Experimental setup

Our experimental platform is a dual-processor 700 MHz Intel Pentium III-based workstation with 512 MB RAM, running Linux 2.4.2. Two SCSI disks connected to the workstation are used for real disk measurements: The IBM Ultrastar 18ES (1998) is a 7,200 RPM disk with 7.6 ms average seek time and 9 GB capacity. The Ultrastar resides on an 80 MB/s SCSI bus hosted by an Adaptec AIC-7896. The Seagate Cheetah X15 (2000) is a 15,000 RPM disk with 3.9 ms average seek time and 18 GB capacity. It is connected to a 1 Gb/s Fibre Channel network (FC-AL) hosted by a QLogic ISP2100. These disks were chosen as reasonable examples of modern high-end disks. Also, validated DiskSim specifications are available for these disks, allowing us to compare the Memulator to real disks. In addition to the models of these disks, we created a simple "10ms Constant Time" model, which always completes requests with 10 ms service time.

For all experiments, 350MB of main memory is pinned for the Memulator's RAM cache, leaving 162MB for the "real" system activity. This memory is pinned even when not using the Memulator in order to equalize the system behavior when comparing the Memulator to real storage devices. The second CPU allows the Memulator application to execute on the local system without taking CPU cycles from our benchmark programs.

To focus on storage performance, we use six artificial workloads: "random or mixed" crossed with "small, uniform, or large." A *random* workload has zero probability of local access or sequential access; request starting locations are uniformly distributed across the storage capacity. A *mixed* workload has 30% probability of "local" access (within 500 LBNs of the previous request) and 20% probability of sequential access. A *small* workload is composed of 8-sector (4 KB) requests, a *large* workload uses 256-sector (128 KB) requests, and a *uniform* workload has uniformly distributed request sizes in intervals of 2 KB over the range [2 KB, 130 KB]. Therefore a “mixed large” workload has some sequential and local accesses, and is composed of 128 KB requests. All workloads are made up of 2,000 requests, of which 67% are reads. This size was chosen to prevent the RAM cache from paging to overflow storage.

We also present results for three application-level benchmarks: the Andrew benchmark [13], the PostMark benchmark [14], and the SSH-build benchmark [25].

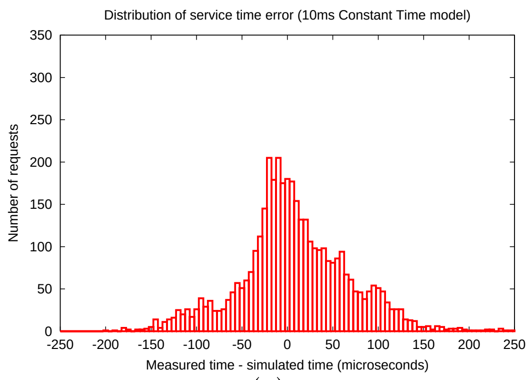
The Andrew file system benchmark has been popular in file system studies since its introduction. Its five phases operate on a source tree of about 70 files (200 KB), with each phase designed to exercise a unique component of the filesystem. Given its age and small size, the Andrew benchmark is not I/O intensive.

PostMark was designed to measure the performance of a file system used for electronic mail, news, and web-based services. It creates a large number of small files, on which a specified number of transactions are performed. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. The transaction types are chosen randomly with consideration given to user definable weights. Our configuration consists of 20,000 transactions on 10,000 files, with a file size of between 10 KB and 20 KB.

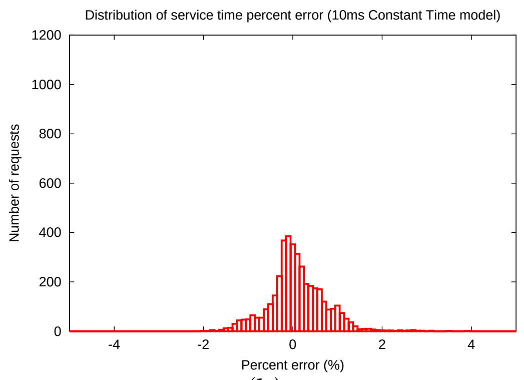
The SSH-build benchmark was constructed as a replacement for the Andrew benchmark. It consists of 3 phases: The unpack phase, which unpacks the compressed tar archive of SSH v3.0.0 (SSH is approximately 2.1MB in size before decompression). This phase stresses metadata operations on files of varying sizes. The configure phase consists of the automatic generation of header files and Makefiles, which involves building various small programs that check the existing system configuration. The build phase compiles, links, and removes temporary files. This last phase is the most CPU intensive, but it also generates a large number of object files and a few executables.

## 5.2 Results

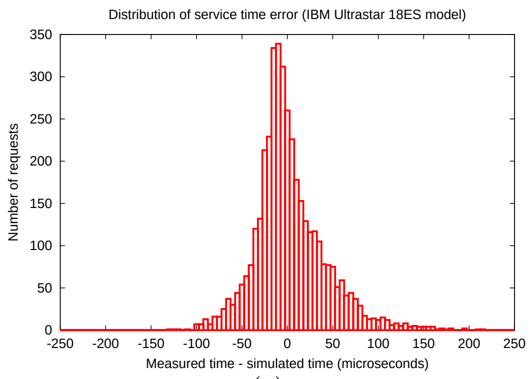
We executed the six artificial workloads against the Memulator to evaluate how closely it comes to perfect timing-accurate emulation. To achieve this, we dynamically generated a series of SCSI requests based on each workload’s characteristics and issued them to the



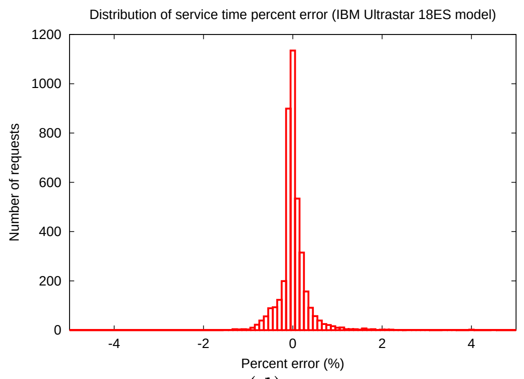
(a)



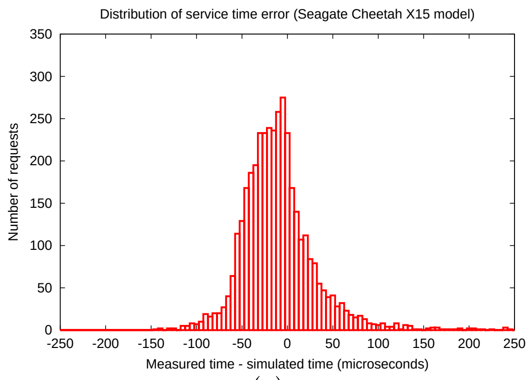
(b)



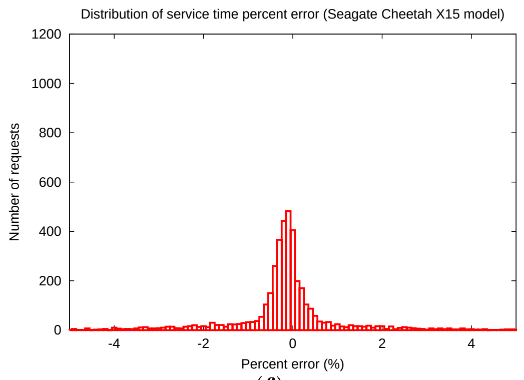
(c)



(d)



(e)



(f)

Figure 6: **Densities of emulation error and percent error.** *Each graph shows the combined results of the "random uniform" and "mixed uniform" workloads, for a total of 4,000 requests. The step in the error densities is 5 μs; in the percent error densities, the step is 0.1%. Percent error is calculated with respect to the simulated request time.*

| | small requests (4 KB) | | uniform (2–130 KB) | | large requests (128 KB) | |
|---|---|---|---|---|---|---|
| | random | mixed | random | mixed | random | mixed |
| **10ms Constant Time model** | | | | | | |
| mean service time | 10,000 $\mu$s | 10,000 $\mu$s | 10,000 $\mu$s | 10,000 $\mu$s | 10,000 $\mu$s | 10,000 $\mu$s |
| mean emulation error | -26.6 $\mu$s | -27.1 $\mu$s | 7.2 $\mu$s | 12.8 $\mu$s | 83.3 $\mu$s | 58.1 $\mu$s |
| mean \|emulation % error\| | 0.28% | 0.28% | 0.49% | 0.46% | 0.87% | 0.84% |
| requests under 2% error | 100% | 100% | 99.1% | 99.3% | 99.3% | 99.5% |
| **IBM Ultrastar 18ES model** | | | | | | |
| mean service time | 11,626 $\mu$s | 9,696 $\mu$s | 22,949 $\mu$s | 21,139 $\mu$s | 35,089 $\mu$s | 33,174 $\mu$s |
| mean emulation error | -10.4 $\mu$s | -4.6 $\mu$s | 9.0 $\mu$s | -1.2 $\mu$s | 18.4 $\mu$s | 48.9 $\mu$s |
| mean \|emulation % error\| | 0.36% | 0.50% | 0.21% | 0.20% | 0.16% | 0.17% |
| requests under 2% error | 99.9% | 96.2% | 99.4% | 99.6% | 100% | 100% |
| **Seagate Cheetah X15 model** | | | | | | |
| mean service time | 6,623 $\mu$s | 5,599 $\mu$s | 9,403 $\mu$s | 8,531 $\mu$s | 11,705 $\mu$s | 10.781 $\mu$s |
| mean emulation error | -13.6 $\mu$s | 5.8 $\mu$s | 41.5 $\mu$s | 17.8 $\mu$s | 58.2 $\mu$s | 60.9 $\mu$s |
| mean \|emulation % error\| | 0.59% | 1.33% | 0.95% | 1.15% | 0.99% | 0.80% |
| requests under 2% error | 99.6% | 99.4% | 97.7% | 98.5% | 99.6% | 99.8% |

Table 2: **Memulator accuracy.** *Each workload represents 2,000 requests as measured at the Linux SCSI generic interface.* Mean service time *is the average request service time reported by the simulation engine.* Mean emulation error *reports the average difference between the measured (emulated) time and the simulated service time of each request. Negative values represent requests that finished more quickly than the simulated time.* Mean |emulation % error| *is the average of the absolute values of percent error of the emulated time for each request with respect to the simulated service time.* Requests under 2% error *shows the percentage of requests completing within 2% of their simulated time.*

| | IBM Ultrastar 18ES | | | Seagate Cheetah X15 | | |
|---|---|---|---|---|---|---|
| | Real disk | Memulator | % error | Real disk | Memulator | % error |
| Andrew | 3.545 s | 3.543 s | -0.06% | 3.538 s | 3.537 s | -0.03% |
| PostMark | 372.7 s | 389.0 s | 4.37% | 14.36 s | 14.31 s | -0.33% |
| SSH unpack | 0.631 s | 0.628 s | -0.48% | 0.627 s | 0.628 s | 0.16% |
| SSH configure | 39.95 s | 40.06 s | 0.28% | 39.10 s | 38.93 s | -0.44% |
| SSH build | 119.3 s | 119.2 s | -0.09% | 119.8 s | 119.0 s | -0.73% |

Table 3: **Application run times using the Memulator vs. real disks.** *Each column shows the average of 10 benchmark runs, except the Postmark numbers for the IBM Ultrastar 18ES (only 3 runs each). Coefficients of variation are below 3%. The Seagate Cheetah X15 Postmark runs are for only 1000 files instead of 10,000 files.*

Memulator through the Linux SCSI generic (SG) interface. The SG interface allows an application to create SCSI requests at the user level, to pass these commands directly to the device driver, to intercept SCSI replies from the driver, and to handle them directly at user level. Timing each request at the SG interface and comparing these to the simulator output enables a detailed request-by-request comparison.

Table 5.1 displays the results, and Figure 6 provides a supplementary view of the uniform workloads. The average |% emulation error| is less than 1.4% in all cases, and over 99% of requests have less than 2% of error. Most errors larger than 2% are only slightly larger. Exceptions fall into two categories: (1) the simulator can take too long to compute a result (see Figure 5(f)), and (2) the emulation program can be context-switched off of the CPU, which occurs for fewer than one request in 1000 in our experiments. Fundamentally, the extra delays from both categories are unbounded, but we have observed only 5-10% inaccuracy from the first and up to 3–4 ms errors from the second.

Having established that Memulator matches its internal simulation timings, we compare application run times over the Memulator vs. over real disks, using the validated DiskSim models of those disks. Our results are shown in Table 5.1. Run times with the Memulator are very close to those with the corresponding real disk. Although these very close matches are comforting, it is important to remember that the Memulator's main responsibility is ensuring fidelity to the model's timing. It is the responsibility of the model's creator to ensure fidelity to the modeled device.

## 6 Memulator-enabled Experiments

This section illustrates the power of timing-accurate storage emulation by describing experiments made possible by the Memulator. These experiments fall into three categories: experiments with firmware modifications, experiments with futuristic devices, and experiments with new storage interfaces.

### 6.1 Changes to existing devices

A long-standing obstacle for most experimental storage researchers is that disk firmware source code is unavailable. This prevents direct experimentation with modifications to

| IBM Ultrastar 18ES | | | |
|---|---|---|---|
| | Default | Zero-latency | Decrease in time |
| PostMark | 389.0 s | 399.0 s | -1.0% |
| SSH unpack | 0.63 s | 0.63 s | 0.0% |
| SSH configure | 40.1 s | 40.0 s | 0.3% |
| SSH make | 119.2 s | 119.7 s | -0.4% |

Table 4: **Exploring a change to disk firmware.** *Here we use timing-accurate storage emulation to add zero-latency access capability to a disk that in reality does not support it. Each data point is the average of three runs of the benchmark, with each coefficient of variation below 3%.*

firmware algorithms, including LBN-to-physical mapping, on-board cache management, prefetching, and scheduling. With the Memulator, this obstacle is partially removed.

To illustrate the new capability, we compare application performance when a disk has zero-latency read support and when it does not. Zero-latency read (a.k.a. read-on-arrival and immediate read) allows the disk firmware to fetch sectors from the media in any order, rather than requiring strictly ascending LBN order. When exactly one track is fetched, zero-latency read support allows the media transfer to begin as soon as the seek is complete; since every sector on the track is desired, the media transfer requires one rotation and there is no rotational latency. Without zero-latency read, the same request would suffer the normal rotational latency before the one rotation of media transfer.

Table 4 shows the performance impact of zero-latency reads on the Postmark and SSH-build benchmarks described in the previous section. Although some disks support zero-latency reads, the IBM Ultrastar 18ES and the Seagate Cheetah X15 do not. For these workloads, this design choice is correct, since there is no significant performance benefit. These workloads all involve mostly small files and background disk writes, and so there is little opportunity to benefit from zero-latency reads. A workload with larger transfers could be expected to benefit.

Although these results may not be interesting, the ability to conduct the experiment is. Enabling full system experimentation may increase the believability of results pertaining to future firmware enhancement proposals.

| | IBM Ultrastar 18ES | MEMS-based storage | Decrease in time |
|---|---|---|---|
| PostMark | 389.0 s | 113.7 s | 70.8% |
| SSH unpack | 0.63 s | 0.64 s | -1.6% |
| SSH configure | 40.1 s | 38.9 s | 3.0% |
| SSH make | 119.2 s | 119.0 s | 0.2% |

Table 5: **MEMS-based storage vs. IBM Ultrastar 18ES.** *Each data point is the average of three runs of the benchmark, with each coefficient of variation below 3%.*

## 6.2 New storage technologies

Microelectromechanical systems (MEMS)-based storage is an exciting new technology that could soon be available in systems. MEMS are very small scale mechanical structures—on the order of 10–1000 $\mu$m—fabricated on the surface of silicon wafers [24]. Using thousands of minute MEMS read/write heads, data bits can be stored in and retrieved from media coated on a small movable media sled [3, 10, 22]. With higher storage densities (260–720 Gbit/in$^2$) and lower random access times (<1 ms), MEMS-based storage devices could play a significant role in future systems.

Fully-functioning MEMS-based storage devices should be available in the next few years, but we would like to explore their role in systems now. The Memulator allows us to do so. Specifically, DiskSim includes the MEMS-based storage device model described by Griffin et al. [10]. Therefore, the Memulator can be configured to emulate these devices, allowing full system experiments with real applications.

Table 5 shows application performance when replacing a disk with MEMS-based storage. For Postmark, MEMS-based storage provides over 3.4$\times$ the transaction throughput (70% reduction in runtime for 20,000 transactions). For SSH-build, minimal performance difference is observed, because the benchmark data stays resident in the file cache and most of the I/Os are background writes. These writes complete faster, but there is no effect on application performance.

## 6.3 Storage interface extensions

A third set of storage designs that would benefit from emulation-based evaluation includes storage interface extensions. Such extensions require that both the host OS and the storage device be modified to utilize a new interface. Not only must the interface be supported, but often the implementations of both sides must change to truly exploit a new interface's potential. Two examples of this arise from recently-promoted mechanisms: freeblock scheduling [16] and eager writing [26].

Freeblock scheduling consists of replacing the rotational latency delays of high-priority disk requests with background media transfers. Since the high-priority data will rotate around to the disk head at the same time, regardless of what is done during the rotational latency, these background media transfers can occur without slowing the high priority requests. It is believed that freeblock scheduling can be accomplished most effectively from within disk firmware. Before they will consider new functionality, however, disk manufacturers want to know exactly what the interface should be and what real application environments will benefit. Since researchers have no access to disk firmware, this creates a chicken-and-egg problem. The Memulator, combined with OS source code (e.g., Linux), enables the interface and application questions to be explored.

Eager writing consists of writing new data to an unused location near the disk head's current location. Such dynamic data placement can significantly reduce service times. As with freeblock scheduling, the best decisions would probably be made from within disk firmware. However, this approach would require the firmware to maintain a mapping table, and it would not benefit from the OS's knowledge of high-level intra-file and inter-file data relationships. A more cooperative interface might allow the host system to direct the disk to write a block to any of several locations (whichever is most efficient); the device would then return the resulting location, which could be recorded in the host's metadata structures. Difficulties would undoubtedly arise with this design, and the Memulator enables OS prototyping and experimentation to flesh them out.

## 7 Summary

This paper describes and promotes timing-accurate storage emulation as a foundation for more thorough evaluation of proposed storage designs. Measurements of our prototype, the Memulator, demonstrate that 99% of its response times are within 2% of their simulator-computed targets. More importantly, the Memulator allows us to run real application benchmarks on real systems equipped with storage components that we cannot yet build, such as disks with firmware extensions and MEMS-based storage.

## References

[1] J. S. Ahn, P. B. Danzig, Z. Liu, and L. Yan. Evaluation of TCP Vegas: emulation and experiment. *ACM SIGCOMM Conference* (Cambridge, MA, 28 August–1 September, 1995). Published as *Computer Communication Review*, **25**(4):185–195. ACM, 1995.

[2] T. E. Anderson, D. E. Culler, and D. A. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, **15**(1):54–64, February 1995.

[3] L. R. Carley, J. A. Bain, G. K. Fedder, D. W. Greve, D. F. Guillou, M. S. C. Lu, T. Mukherjee, S. Santhanam, L. Abelmann, and S. Min. Single-chip computers with microelectromechanical systems-based magnetic memory. *Journal of Applied Physics*, **87**(9):6680–6685, 1 May 2000.

[4] P. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio file cache: surviving operating system crashes. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 1–5 October 1996). Published as *SIGPLAN Notices*, **31**(9):74–83, 1996.

[5] K. Fall. Network emulation in the Vint/NS simulator. *IEEE Symposium on Computers and Communications* (Red Sea, Egypt, 6–8 July 1999), pages 244–250, 1999.

[6] G. R. Ganger. Generating representative synthetic workloads: an unsolved problem. *International Conference on Management and Performance Evaluation of Computer Systems* (Nashville, TN), pages 1263–1269, 1995.

[7] G. R. Ganger and Y. N. Patt. Using system-level models to evaluate I/O subsystem designs. *IEEE Transactions on Computers*, **47**(6):667–678, June 1998.

[8] G. R. Ganger, B. L. Worthington, R. Y. Hou, and Y. N. Patt. Disk arrays: high-performance, high-reliability storage systems. *IEEE Computer*, **27**(3):30–36, March 1994.

[9] G. R. Ganger, B. L. Worthington, and Y. N. Patt. *The DiskSim simulation environment version 1.0 reference manual*, Technical report CSE–TR–358–98. Department of Computer Science and Engineering, University of Michigan, February 1998.

[10] J. L. Griffin, S. W. Schlosser, G. R. Ganger, and D. F. Nagle. Modeling and performance of MEMS-based storage devices. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Santa Clara, CA, 17–21 June 2000). Published as *Performance Evaluation Review*, **28**(1):56–65, 2000.

[11] J. L. Griffin, S. W. Schlosser, G. R. Ganger, and D. F. Nagle. Operating system management of MEMS-based storage devices. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 227–242. USENIX, 2000.

[12] D. Hitz. *An NFS file server appliance*. Technical report. Network Appliance, August 1993.

[13] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, **6**(1):51–81, February 1988.

[14] J. Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.

[15] D. Kotz, S. B. Toh, and S. Radhakrishnan. *A detailed simulation model of the HP 97560 disk drive.* Technical report PCS–TR94–220. Department of Computer Science, Dartmouth College, July 1994.

[16] C. R. Lumb, J. Schindler, G. R. Ganger, D. F. Nagle, and E. Riedel. Towards higher disk head utilization: extracting free bandwidth from busy disk drives. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 87–102. USENIX Association, 2000.

[17] B. D. Noble, M. Satyanarayanan, G. T. Nguyen, and R. H. Katz. Trace-based mobile network emulation. *ACM SIGCOMM Conference* (Cannes, France, 14–18 September 1997), pages 51–61, 1997.

[18] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM. Transactions on Modeling and Computer Simulation*, **7**(1):78–103. ACM, January 1997.

[19] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO, 3–6 December 1995). Published as *Operating Systems Review*, **29**(5), 1995.

[20] C. Ruemmler and J. Wilkes. UNIX disk access patterns. *Winter USENIX Technical Conference* (San Diego, CA, 25–29 January 1993), pages 405–420, 1993.

[21] S. W. Schlosser, J. L. Griffin, D. F. Nagle, and G. R. Ganger. Designing computer systems with MEMS-based storage. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 12–15 November 2000). Published as *Operating Systems Review*, **34**(5):1–12, 2000.

[22] P. Vettiger, M. Despont, U. Drechsler, U. Dürig, W. Häberle, M. I. Lutwyche, H. E. Rothuizen, R. Stutz, R. Widmer, and G. K. Binnig. The “Millipede”—more than one thousand tips for future AFM data storage. *IBM Journal of Research and Development*, **44**(3):323–340, 2000.

[23] R. Y. Wang, D. A. Patterson, and T. E. Anderson. Virtual log based file systems for a programmable disk. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22–25 February 1999), pages 29–43. ACM, 1999.

[24] K. D. Wise. Special issue on integrated sensors, microactuators, and microsystems (MEMS). *Proceedings of the IEEE*, **86**(8):1531–1787, August 1998.

[25] T. Ylonen. SSH — secure login connections over the Internet. *USENIX Security Symposium* (San Jose, CA, 22–25 July 1996). USENIX Association, 1996.

[26] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading capacity for performance in a disk array. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 243–258. USENIX Association, 2000.