Timing-accurate Storage Emulation

John Linwood Griffin, Jiri Schindler, Steven W. Schlosser,
John S. Bucy, Gregory R. Ganger
Carnegie Mellon University

Abstract

Timing-accurate storage emulation fills an important gap 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).

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, 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 determines 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 a real system.

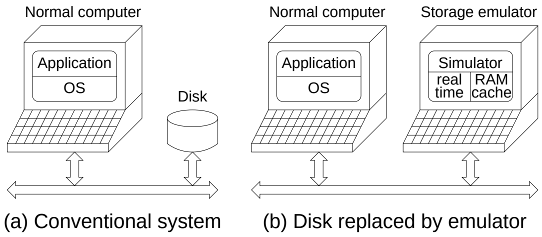

Figure 1: ***A system with (a) real storage or (b) emulated storage.*** *The emulator transparently replaces storage devices in a real system. By reporting request completions at the correct times, the performance of different devices can be mimicked, enabling full system-level evaluations of proposed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

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. Our original goal was thorough evaluation of operating system algorithms for not-yet-existing MEMS-based storage devices [11, 12]—this led to the prototype's name: *Memulator*. The Memulator integrates the DiskSim simulator [10], 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 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 as with 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 [9]. 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 workload generation is still an open problem, I/O traces ignore system feedback effects [9], and available traces are often out-of-date—in fact, many storage researchers still rely on the decade-old "HP traces" from 1992 [21]. 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 [3, 17, 19]. Under 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

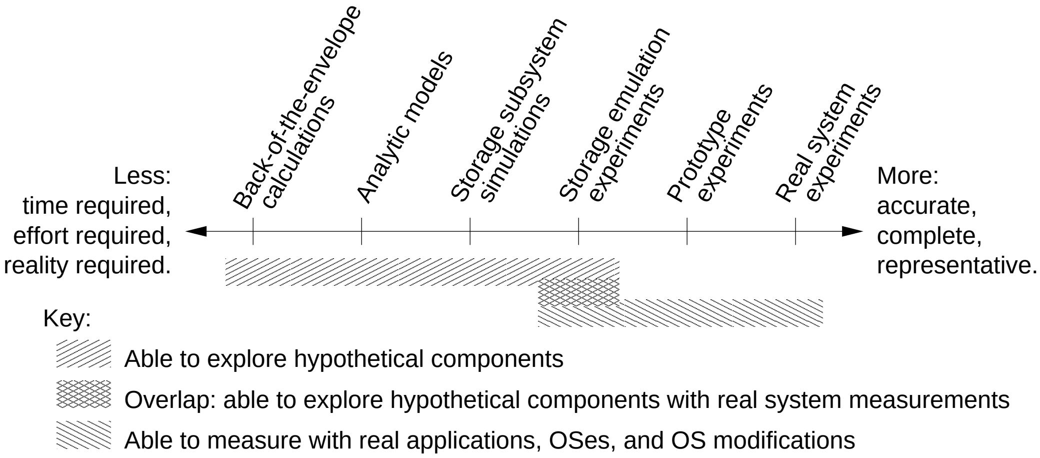

Figure 2: ***Storage performance evaluation techniques.*** *This illustration linearizes techniques across a spectrum from the ⟨quickest,easiest,most flexible⟩ to the most ⟨accurate,complete,representative⟩. In this spectrum, storage emulation provides the unique ability to explore nonexistent storage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

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 10 GHz CPUs) can be evaluated [20, 22]. 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 [13] 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, 6, 18]. 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 emulation, 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 the evaluation of eager writing by Wang et al. [25]. 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 RAM disk, and arranged (using the `sleep()` system call) 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 [5, 8]. Although this is unacceptable for 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.

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 three tasks: communications management (the storage interface), timing management (the simulation engine and timing loop), and data management (the RAM cache and overflow storage).

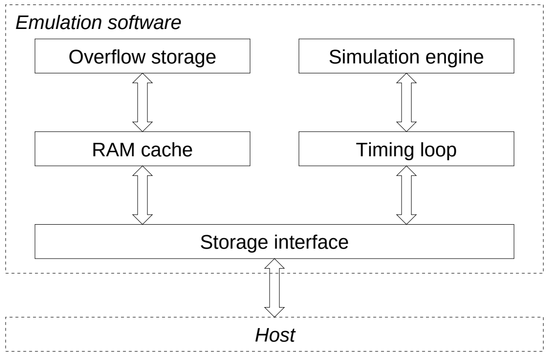

Figure 3: ***Emulation software internals.*** *The five components inside the "storage emulation software" box comprise the three primary emulator tasks: communications management (the storage interface), timing management (the simulation engine and timing loop), and data management (the RAM cache and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

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 support control requests that return information about the emulated drive such as its capacity, status, or error condition. 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.

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. The emulator-based evaulation of eager writing [25] used a disk simulator by Kotz et al. [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 wall clock time, and the timing loop calls back into the simulator immediately. When the 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. An obvious 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.

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 swapping 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 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, 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 simply 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 (*local* emulation), the device driver is modified to communicate directly with the emulation software rather than with real storage components. Although this does involve some modifications to the system under test, they are restricted to the device driver. In the second (*remote* emulation), 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 as 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 3-step interface between the emulation software and the rest of the system.

Step 1: Send the 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 initiates the data transfer. 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 buffers in its RAM cache directly to the modified device driver. The driver then copies data between these userspace buffers and the appropriate kernel buffers. A remote emulator sends data over the bus to the host.

Step 3: Send the reply to the device driver. The emulation software waits until the request service time as determined in Step 1 elapses. At this point, 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.

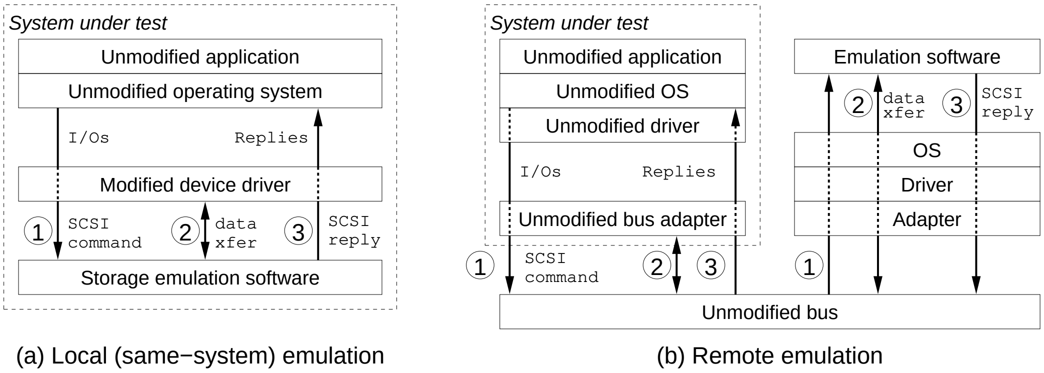

Figure 4: ***Communication paths when emulation is run (a) locally or (b) remotely.*** *When run locally, emulation software communicates directly with a modified device driver in the kernel. Under remote emulation, all modifications take place outside the system under test, eliminating any impact of the emulation overheads. The three steps are described in Section 3.2.*

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 some 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 can mitigate this 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. An advantage of this is that it allows emulation of devices "connected" to very fast local buses (for example, the PCI or system bus) or even emulation of the interconnect itself. 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 Implementation of the Memulator

This section describes the implementation of the *Memulator*, a prototype timing-accurate storage emulator. The emulation software runs as a user-level application and communicates with the host via a modified SCSI device driver. The Memulator can be run as either a local or remote emulator.

User-level emulation software. In the Memulator, user-level emulation software does the core work of timing-accurate storage emulation. It interprets requests, retains stored data, simulates device timings, and sends replies after the correct delays. This component is common to both the local and remote Memulator.

Timings are computed by DiskSim, which is an event-driven storage simulator [10]. The Memulator interacts with DiskSim via its "external control mode," in which external software (the timing loop) calls into DiskSim, specifies how far the simulation time should proceed before control is returned, and is then told when the next DiskSim-internal event should happen. The timing loop keeps the simulation time in close proximity to the real-world clock as given by the processor's time stamp counter.

Main memory is used to hold data written by previous requests. The Memulator's RAM cache is allocated and pinned in-core in its entirety during initialization using the `malloc()` and `mlock()` system calls. The operating system's resource limits may need to be adjusted to allow the pinning of a large memory region. The Memulator does not currently support overflow storage, so the working set of each experiment is limited to the cache's capacity.

When invalid opcodes, out-of-range requests, or invalid target/LUN pairs are received, the Memulator's storage interface generates the appropriate sense code and immediately returns an error condition. Table 1 lists the SCSI commands supported by the Memulator. These commands are sufficient to allow Linux or FreeBSD to mount and use Memulator devices as SCSI disks.

Local emulation. The local version of the Memulator runs on the Linux 2.4 operating system. When used for local emulation, the user-level emulation software runs on the system under test as illustrated in Figure 4(a).

The modified device driver is a low-level component in the Linux SCSI subsystem. The driver accepts SCSI requests (`Scsi_Cmnd` structures) from the Linux kernel via the standard mid-to-low-level `queuecommand()` interface and passes them 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 device driver via modified system calls on the special character device file `/dev/memulator`. The storage interface uses the `poll()` system call to discover that a new request has arrived at the driver. It then uses the `read()` system call to transfer the 6-, 10-, or 12-byte SCSI command, the target device number, the logical unit number, and a unique request identifier. Following this transfer the timing loop immediately prepends a REQUEST ARRIVAL event to the DiskSim event queue. The arrival timestamp is skewed approximately 25μs into the past to account for communications overheads; this value was determined empirically for our experimental setup. Once the newly arrived request is enqueued, the storage interface immediately directs the device driver to copy the requested data between the user and kernel memory buffers. The emulation software later uses the `write()` system call to notify the device driver when it determines the request is complete.

Table 1: ***Required SCSI command support.*** *Support for the upper six commands is necessary for an emulator to interact with a Linux 2.4 host. All nine commands must be supported when communicating with a FreeBSD 4.4 host.*

Command	Function
READ (6 and 10)	Read data from device
WRITE (6 and 10)	Write data to device
TEST UNIT READY	Check for device online
INQUIRY	Get device parameters
READ CAPACITY	Get device size in sectors
REQUEST SENSE	Get details of last error
MODE SENSE	Configure device
WRITE AND VERIFY	Verify data on device
SYNCHRONIZE CACHE	Flush device cache

Remote emulation. The remote version of the Memulator runs on the FreeBSD 4.4 operating system. When used for remote emulation, the Memulator runs entirely on a separate computer system. Both the host and remote systems are connected to a shared storage interconnect as illustrated in Figure 4(b).

Remote emulation requires hardware support in the bus adapter to act as a target in order to receive commands from an initiator. The operating system must also handle incoming requests and direct them to the user-level emulation software. This support is provided by FreeBSD's CAM subsystem when used with certain SCSI or Fibre Channel cards. The storage interface communicates with a modified target mode device driver in much the same manner as described for the local version of the Memulator. For remote emulation experiments the arrival timestamp is skewed approximately 120μs into the past.

Alternative implementations of a remote emulator could leverage storage networking protocols such as iSCSI or run on dedicated custom hardware connected to the PCI or system buses of the host system.

5 Memulator validation

This section presents three evaluations of the Memulator. First, we show that the upper bound of performance for our setup is sufficient to meet the requirements of the devices we model. Second, we show that the Memulator accurately reflects the timings of a simulated storage device. Third, we show that this timing-accuracy can translate into accurate emulation of a real storage device.

5.1 Experimental setup

Local emulation experiments are performed on a 700 MHz dual-processor Intel Pentium III-based workstation with 2 GB RAM running Linux 2.4.2. The second CPU is used to reduce interference between the emulation software and the regular workload. During all experiments, 1,792 MB of main memory is pinned as the Memulator's RAM cache, leaving 256 MB for the "real" system activity. Unless otherwise specified, the experiments in this paper are run under local emulation.

For remote emulation, the Memulator runs on a single-processor workstation with 2 GB RAM and FreeBSD 4.4. The host system is a single-processor workstation with 256 MB RAM running either Linux or FreeBSD as noted below. The host and remote systems are connected by an 80 MB/s SCSI bus via Adaptec AHA-29160 adapters.

The disk used for comparison is the Seagate Cheetah X15, a 15,000 RPM disk with 3.9 ms average seek time and 18 GB capacity. It is connected to a 1 Gbit/s Fibre Channel network (FC-AL) hosted by a QLogic ISP2100.

This disk was chosen as a reasonable example of a modern high-end disk. Validated DiskSim specifications are available [7] for this disk, allowing us to configure the Memulator using validated parameters for the Cheetah X15.

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 1,000 I/O requests, of which 67% are reads.

We also present results for the PostMark benchmark [14]. 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. The benchmark parameters in these experiments specify 20,000 transactions on 10,000 files, with a file size of between 10 KB and 20 KB.

5.2 Implementation performance

To determine the fastest device that the Memulator can emulate, we configured it to reply with request completions immediately after the data transfer phase. By removing the timing component, all that remains is the overhead required for emulation.

We measure request rate and bandwidth for both local and remote configurations. Request rate is measured by issuing 2^{10} one-sector read requests and dividing the number of requests by the elapsed time. Bandwidth is measured by issuing 2^{10} 1024-sector read requests and dividing the total bytes transferred by elapsed time. System-level caching is disallowed; all requests are synchronously issued through the Linux SCSI generic (SG) interface in local emulation and the FreeBSD direct access interface during remote emulation.

The results for local and remote emulation are shown in Table 2, along with the required performance values for MEMS-based storage [11] and the Seagate Cheetah X15 [23]. Both the local and remote Memulator configurations achieve the required performance threshold for both devices along both axes.

5.3 Memulator accuracy

To evaluate how closely the Memulator comes to perfect timing-accurate emulation, we execute the six artificial workloads against both the Memulator and against standalone DiskSim. We run these workloads against the Memulator by generating a series of SCSI requests based on each workload's characteristics and issuing the requests to the Memulator through the Linux SCSI generic (SG) interface. The SG interface allows an application to create SCSI requests at the user level, pass these commands directly to the device driver, receive replies from the driver, and handle them directly at user level. We measure the Memulator's accuracy by taking arrival and completion timestamps inside the device driver, then comparing these times to the per-request times reported by standalone DiskSim. If these times match, the kernel is seeing exactly the performance intended.

Table 3 displays the results, and Figure 5 provides a supplementary view of the uniform workloads. The average $|\%$ emulation error$|$ (that is, the average per-request error, independent of whether the request completed too fast or too slow) is less than 0.33% in all cases, and over 99% of all requests have less than 2% 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, and (2) the emulation program can be context-switched off of the CPU. These exceptions occurs for about one request in 1000 in the 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.

Table 2: ***Upper bound of Memulator performance.*** *The local and remote emulation values are measured with the Memulator configured to return data and replies as quickly as possible. The remote emulation experiments are run using a FreeBSD host. The MEMS-based storage [11] and Seagate Cheetah X15 [23] values represent peak bandwidth and average request rate for random I/O.*

	Request rate	Bandwidth
Available performance		
Local emulation	22,468 req/s	84.5 MB/s
Remote emulation	8,883 req/s	103.5 MB/s
Required performance		
MEMS-based storage	1,422 req/s	76 MB/s
Cheetah X15 disk	244 req/s	49 MB/s

Table 3: ***Memulator accuracy.*** *Each workload represents 1,000 requests as measured inside the device driver and inside DiskSim. "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 1% error" shows the percentage of requests completing within 1% of their simulated 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	-0.65μs	-0.49μs	0.93μs	0.75μs	2.36μs	1.42μs
mean |emulation % error|	0.01%	0.01%	0.04%	0.02%	0.05%	0.05%
requests under 1% error	100%	100%	99.9%	100.0%	100.0%	99.9%
Cheetah X15 model						
mean service time	$6,509\mu$s	$5,568\mu$s	$9,116\mu$s	$8,410\mu$s	$11,301\mu$s	$10,154\mu$s
mean emulation error	5.33μs	10.87μs	0.92μs	-0.19μs	-10.08μs	-9.70μs
mean |emulation % error|	0.11%	0.33%	0.07%	0.08%	0.09%	0.11%
requests under 1% error	99.5%	92.4%	99.9%	99.1%	100%	100%
MEMS-based storage model						
mean service time	$1,057\mu$s	$1,049\mu$s	$2,001\mu$s	$1,957\mu$s	$2,846\mu$s	$2,857\mu$s
mean emulation error	-0.21μs	0.16μs	-0.59μs	-0.50μs	-1.54μs	-0.01μs
mean |emulation % error|	0.13%	0.16%	0.13%	0.14%	0.13%	0.11%
requests under 1% error	100%	99.4%	100%	98.9%	100%	100%

Table 4: ***Application run times using the Memulator vs. using a real disk.*** *Both the local and the remote Memulator faithfully reproduce the performance of the disk under the PostMark benchmark, but there is some interference with application performance in the local emulation case. PostMark was configured to use 10,000 files between 10 KB and 20 KB in size, with 20,000 transactions. Each data point is the average of ten runs of the benchmark. The remote emulation experiments are run using a Linux host.*

	Real Cheetah X15	Local emulation	Remote emulation
PostMark			
run time	78.422 s	74.523 s	78.532 s
standard deviation	0.375	0.618	0.244
percent error from real disk	—	4.97%	0.14%

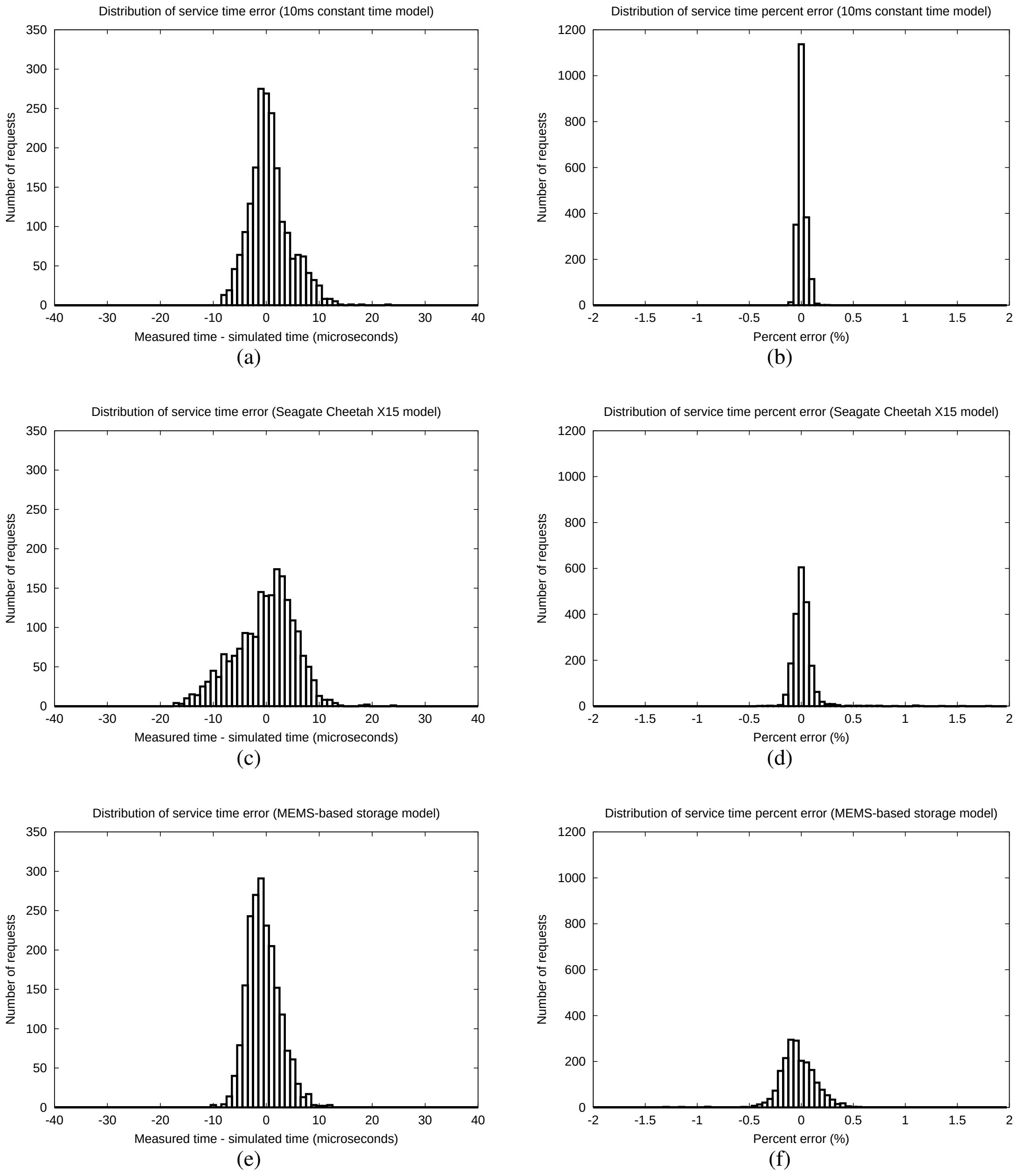

Figure 5: ***Densities of emulation error and percent error.*** *Emulation error is defined as the difference between the time reported by the emulator and that reported by the simulator alone running the same workload. This error shows the timing discrepancy caused by variation in the overheads of emulation, such as passing commands and data to the emulation software. A perfect emulator would introduce no discrepencies and the times would match exactly. The Memulator is shown to introduce only minor discrepencies compared to DiskSim running alone. Each graph shows the combined results of the "random uniform" and "mixed uniform" workloads, for a total of 4,000 requests. Percent error is calculated with respect to the simulated request time. Bin sizes are 1*µs *in the service time graphs and 0.05% in the percent error graphs.*

Table 5: ***Exploring a change to disk firmware.*** *Here we use timing-accurate storage emulation to add zero-latency access capability to a disk (the Seagate Cheetah X15) that in reality does not support it. Each data point is the average of ten runs of the benchmark.*

	Default	Zero-latency	Decrease in time
PostMark			
run time	74.523 s	74.469 s	0.1%
std. dev.	0.618	0.783	—

Table 6: ***Exploring technology trends.*** *These results show the effect of scaling the rotation speed of the Seagate Cheetah X15 to 30,000 RPM. Each data point is the average of ten runs of the benchmark.*

	15,000 RPM	30,000 RPM	Decrease in time
PostMark			
run time	74.523 s	66.215 s	11.1%
std. dev.	0.618	0.651	—

5.4 Comparison with real disks

Having established in Section 5.3 that the Memulator matches its internal simulation timings, we compare application run times using the Memulator vs. using real disks. The results are shown in Table 4. In this experiment, we see the advantage of using remote emulation rather than local emulation. Interactions between the "real" application/OS activity and local emulation activity cause the benchmark runtime to be off by 5%. With remote emulation, on the other hand, benchmark performance is almost identical (within 0.14%) whether using the disk or the Memulator. Although the close match for remote emulation is comforting, it is important to remember that the Memulator's main responsibility is to ensure 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 modified disks, experiments with futuristic devices, and experiments with storage interface extensions.

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 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 at most one rotation. Without zero-latency read, the same request would suffer the normal rotational latency before the one rotation of media transfer.

Table 5 shows the performance impact of zero-latency reads on the PostMark benchmark described in the previous section. Although some disks support zero-latency reads, the Cheetah X15 does not. This design choice is correct for PostMark: as the workload involves mostly small files and background disk writes, there is little opportunity to benefit from zero-latency reads. A workload with larger transfers could be expected to benefit.

Although this particular result 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.

In addition to firmware and algorithmic changes, timing-accurate storage emulation enables experiments reflecting hardware and technology changes. For example, Table 6 shows the performance impact of doubling the Cheetah X15's rotational speed. Despite reducing rotational latency by half and doubling the media transfer rate, this hardware upgrade results in only an 11.1% improvement for PostMark.

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 10–1000μm—fabricated on the surface of silicon chips [26]. 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 [4, 11, 24]. With higher storage densities (260–720 Gbit/in^2) and lower random access times (under 1 ms), MEMS-based storage devices could play a significant role in future systems.

The Memulator allows us to explore the impact of using MEMS-based storage in existing computer systems, even

Table 7: ***MEMS-based storage vs. Seagate Cheetah X15.*** *These results compare the runtime of the PostMark benchmark on a Cheetah X15 disk and a MEMS-based storage device. Each data point is the average of ten runs of the benchmark.*

	Cheetah X15	MEMS-based	Decrease in time
PostMark			
run time	74.523 s	51.420 s	31.0%
std. dev.	0.618	1.678	—

though the devices themselves are several years from production. We configured the Memulator to use the G2 device described by Schlosser et al. [22].

Table 7 compares the performance of PostMark running on the Cheetah X15 and on MEMS-based storage. Although the average response time of the disk was five times greater than the MEMS-based storage device (7.91 vs. 1.59 ms), we observed only a 31% decrease in overall runtime when using MEMS-based storage. This is because of the relatively small dataset (only 10,000 10–20 KB files) and the aggressive writeback caching performed by the Linux host system. This caching masks much of the benefit of the faster I/O times of MEMS-based storage. For workloads with larger data sets or synchronous writes (e.g., transaction processing), the overall system performance improvements would be greater. The performance with MEMS-based storage approaches our setup's minimum runtime of 48.512 s, which we measured by rerunning the experiment with the Memulator configured to respond to all I/O requests immediately.

6.3 Storage interface extensions

A third set of storage designs that would benefit from emulation-based evaluation involves 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-proposed mechanisms: freeblock scheduling [16] and eager writing [25].

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.

Acknowledgments

We thank our shepherd Darrell Long, the anonymous reviewers, Andy Klosterman, and Liddy Shriver for helping us refine this paper. We thank the members and companies of the Parallel Data Consortium (including EMC, HP, Hitachi, IBM, Intel, LSI Logic, Lucent, Network Appliances, Panasas, Platys, Seagate, Snap, Sun, and Veritas) for their interest, insights, feedback, and support. We thank IBM and Intel for hardware grants supporting our research efforts. We thank Justin Gibbs for assistance in configuring target mode under FreeBSD. John Griffin is supported in part by a National Science Foundation Graduate Fellowship. Steve Schlosser is supported by an Intel Graduate Fellowship.

References

[1] Jong Suk Ahn, Peter B. Danzig, Zhen Liu, and Limin 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] Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, **15**(1):54–64, February 1995.

[3] M. D. Canon, D. H. Fritz, J. H. Howard, T. D. Howell, M. F. Mitoma, and J. Rodriguez-Rosell. A virtual machine emulator for performance evaluation. *Communications of the ACM*, **23**(2):71–80. ACM, February 1980.

[4] L. Richard Carley, James A. Bain, Gary K. Fedder, David W. Greve, David F. Guillou, Michael S. C. Lu, Tamal Mukherjee, Suresh Santhanam, Leon Abelmann, and Seungook Min. Single-chip computers with microelectromechanical systems-based magnetic memory. *Journal of Applied Physics*, **87**(9):6680–6685, 1 May 2000.

[5] Peter Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David 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.

[6] Kevin 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.

[7] Greg Ganger and Jiri Schindler. Database of validated disk parameters for DiskSim. http://www.ece.cmu.edu/~ganger/disksim/diskspecs.html.

[8] Gregory R. Ganger and Yale N. Patt. Metadata update performance in file systems. *Symposium on Operating Systems Design and Implementation* (Monterey, CA), pages 49–60. Usenix Association, 14–17 November 1994.

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

[10] Gregory R. Ganger, Bruce L. Worthington, and Yale 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.

[11] John Linwood Griffin, Steven W. Schlosser, Gregory R. Ganger, and David 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.

[12] John Linwood Griffin, Steven W. Schlosser, Gregory R. Ganger, and David 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.

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

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

[15] David Kotz, Song Bac Toh, and Sriram 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] Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, David F. Nagle, and Erik 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] Peter S. Magnusson, Fredrik Dahlgren, Håkan Grahn, Magnus Karlsson, Fredrik Larsson, Fredrik Lundholm, Andreas Moestedt, Jim Nilsson, Per Stenström, and Bengt Werner. SimICS/sun4m: a virtual workstation. *Annual USENIX Technical Conference* (New Orleans, LA, 15–19 June 1998), pages 119–30. USENIX Association, 1998.

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

[19] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen 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.

[20] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop 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.

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

[22] Steven W. Schlosser, John Linwood Griffin, David F. Nagle, and Gregory 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.

[23] Seagate Technology, Inc. *Cheetah X15 FC Disc Drive ST318451FC/FCV Product Manual, Volume 1*, Publication number 83329486, Rev. A, June 1, 2000.

[24] 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.

[25] Randolph Y. Wang, David A. Patterson, and Thomas 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.

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