TIMING-ACCURATE STORAGE EMULATION:
EVALUATING HYPOTHETICAL STORAGE COMPONENTS IN REAL COMPUTER SYSTEMS

John Linwood Griffin
September 13, 2004
Technical Report CMU–PDL–04–108

Department of Electrical and Computer Engineering
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, Pennsylvania, 15213-3890
United States of America

A dissertation, submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the field of Electrical and Computer Engineering.

Thesis Committee:

Prof. Gregory R. Ganger, chair
Prof. David R. O'Hallaron
Prof. Remzi H. Arpaci-Dusseau, University of Wisconsin—Madison
Dr. Erik Riedel, Seagate Technology
Dr. Leendert van Doorn, IBM Corporation

Keywords: real-time simulation, performance evaluation, virtualization, active disks, interfaces.

Dedication

To my mentors and teachers over the past quarter-century, especially my father,
in thanks for your gifts of encouragement, nurturing, guidance, and freedom.

ABSTRACT

Timing-accurate storage emulation offers a unique performance evaluation capability: the flexibility of simulation with the reality of experimental measurements. This 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 (or other computational model) of that component. This allows simulated storage components to be plugged into real systems, which can then be used for complete, application-based experimentation. Additionally, timing-accurate storage emulation offers the opportunity to investigate more expressive interfaces between storage and computer systems, permitting forays into the space of hypothetical device functionalities without the difficulties of developing and supporting extensively nonstandard or novel interface actions in prototype or production systems. This dissertation identifies that there is a current and pressing need for a new storage evaluation technique, discusses design issues for achieving accurate per-request service times in a timing-accurate storage emulator, and demonstrates that it is feasible to construct and use such an emulator for interesting system-level experimentation.

We built a functional timing-accurate storage emulator and explored its use in experiments involving models of existing storage products, experiments evaluating the potential of nonexistent storage components, and experiments evaluating interactions between modified computer systems and expanded storage device functionality. To experiment with existing and hypothetical storage components in computer systems, we configured our emulator with device models representing a currently-available production disk drive, a hypothetical 50,000 RPM disk drive, and a hypothetical MEMS-based storage device, and executed three application-level workloads against these emulated models. To explore system architectures with expanded device functionality, we applied the principles of timing-accurate storage emulation in an investigation into storage-based intrusion detection systems. This experimentation demonstrates that our emulator accurately reflects the performance of modeled devices, demonstrates the feasibility of including intrusion detection capabilities into a standalone processing-enhanced disk drive, and demonstrates that extensions to existing storage communications paths may be used to transmit and receive information regarding the configuration and operational status of such an enhanced device.

ACKNOWLEDGEMENTS

This dissertation is the result of the great deal of collaboration, guidance, and friendship I have enjoyed over the years, both before and during my time at Carnegie Mellon. I am grateful to a great number of folks for both the big things and the little things—many of which I'm sure I'm not even aware of—that have supported my wondrous journey to and through graduate school.

Dr. Greg Ganger has been and continues to be an outstanding advisor for me and my colleagues. I cannot say enough about his visionary thinking, the patience with which he develops his students, his exemplary example and openness about the rigors and rewards of academic life, and his willingness to go above and beyond to ensure his students succeed both professionally and personally. It has been a privilege to study under his tutelage. Greg, thank you.

Dr. Richard Chapman at Auburn identified early that I was interested in doctoral studies and invited me to work with his group as an undergraduate assistant to learn what this whole "research" thing is all about. This experience was invaluable as I transitioned to life as a graduate student.

I have been in the enviable position of working directly with my dedicated and capable good friends and research partners on much of the work represented herein. **John Bucy** brought his excellent software programming and aluminum foil-based hardware debugging skills to the table in the development of the communications manager for our remote emulation prototype.
Dr. Jiri Schindler and **Dr. Steve Schlosser**, two of my greatest friends, were inspirational in their dedicated and structured pursuit toward developing an understanding of the potential of track-aligned extents and MEMS-based storage respectively in computer systems.
Adam Pennington deftly served as the driving force behind our investigations of the utility and timeliness of integrating intrusion detection functionality into storage devices. I benefitted greatly from my collaborations with **Dr. Dave Nagle** and **Dr. Rick Carley** on MEMS-based storage; **Chris Lumb** on track-aligned extents; **Garth Goodson**, **Craig Soules**, and **John Strunk** on intrusion detection; **Deepa Choundappan** and **Nithya Muralidharan** on block-based intrusion detection systems; and **Brian Railing** on data management for emulation.

Joan Digney, an accomplished graphic designer, volunteered to transform my crude hand-sketched figures for this dissertation and other research materials into veritable works of art. This was a process akin to transforming caterpillars into butterflies; I remain awe-struck with what she can accomplish from a page filled with otherwise illegible scrawl. **Justin Gibbs**, **Nate Lawson**, **Kenneth Merry**, and **Chuck Tuffli** all provided timely and comprehensive answers to our development and implementation questions posted to the `freebsd-scsi` mailing list.
Jeff Fellin and **Mike Flaster** at Bell Labs, **Dr. Fred Douglis** at IBM Research and **Dr. Arif Merchant** at Hewlett-Packard Labs provided valuable insight into design issues involved with timing-accurate storage emulation.

I had the excellent fortune of acquiring a stellar and interestingly diverse thesis committee, containing as it did 25% more wisdom than most committees. In addition to my being grateful for their expert guidance and suggestions concerning my research directions, I am extremely thankful to **Dr. Remzi Arpaci-Dusseau**, **Dr. Dave O'Hallaron**, **Dr. Erik Riedel**, and **Dr. Leendert van Doorn** for taking an exceptionally focused interest in my future and offering me many practical and thoughtful nuggets of wisdom from their own highly successful careers.

I am especially grateful to the organizations that have provided financial support during my academic career. As regards the work in this dissertation, I am glad to recognize the **National Science Foundation**, **Intel Corporation**, the **Pennsylvania Infrastructure Technology Alliance**, and the **Department of Electrical and Computer Engineering** at Carnegie Mellon for having directly supported my research through fellowship awards and grants. I am also glad to recognize the members and companies who have been part of the Parallel Data Consortium between 1998 and 2004—including **3Com Corporation**, **Bell Labs/Lucent Technologies**, **CLARiiON Array Development** (formerly Data General), **Compaq Corporation**, **EMC Corporation**, **Engenio Information Technologies, Inc.** (formerly LSI Logic and Symbios Logic), **Hewlett-Packard Labs**, **Hitachi Global Storage Technologies** (HGST), **Hitachi, Ltd.**, **IBM Corporation**, **Infineon Technologies** (formerly Siemens Microelectronics), **Intel Corporation**, **Microsoft Corporation**, **MTI Technology Corporation**, **Network Appliance, Inc.**, **Novell, Inc.**, **Oracle Corporation**, **Panasas, Inc.**, **Platys Communications**, **Procom Technology**, **Seagate Technology**, **Snap Appliances** (formerly Quantum Corporation), **Storage Technology** (Storage Tek), **Sun Microsystems, Inc.**, **VERITAS Software Corporation**, and **Wind River Systems, Inc.**—as well as the **Air Force Research Laboratory** and the **Defense Advance Research Projects Agency**, for having supported my research both through advice given during frequent visits to our laboratory and through generous contributions to our laboratory.

Others have discussed how amazing it is to spend over 20% of one's life affiliated with the Parallel Data Laboratory at Carnegie Mellon [7, 33, 39, 69, 85, 124, 129, 139, 144, 164, 178]. I can only reiterate that the personal-development-oriented, collaborative teamwork spirit found here is nothing short of fantastic. Particular credit is deserved by **Dr. Garth Gibson** for having worked ceaselessly to provide the raw materials that ensured the laboratory would succeed. I am very much appreciative of my colleagues in the PDL—especially those mentioned above and **Michael Abd-El-Malek**, **Dave Friedman**, **Charles Hardin**, **James Hendricks**, **Jon Kliegman**, **Andy Klosterman**, **Mike Mesnier**, **David Petrou**, **Brandon Salmon**, **Eno Thereska**, **Niraj Tolia**, **Dr. Ted Wong**, **Jay Wylie**, and **Shuheng Zhou**—who made my time with them a daily enjoyment and who were instantly available whenever I had a question or needed help.

Everywhere I've been I've been blessed to work with absolutely top-notch departmental advisors and administrative assistants. My favorite ladies—**Judy Aull**, **Gloria Bailey**, **Cheryl Matheny**, **Sandra Simmons**, and **Sherie Vandervoort** at Auburn, and **Elaine Lawrence**, **Karen Lindenfelser**, **Lynn Philibin**, and **Linda Whipkey** at Carnegie Mellon—never seemed to mind my popping into their office unannounced to wile away the afternoons, and each still brightens my day every time she walks into the room. One starts to wonder why I ever wanted to graduate, since every move forced me to leave them behind. Additionally, the computing support staff in the Department of Electrical and Computer Engineering at Carnegie Mellon is world-class, with well-deserved kudos to **Brandon Allbery**, **Lou Anschuetz**, **Jim McKinney**, and **Tim Talpas** for being available to diagnose and fix network problems at 3:00 AM on a holiday.

Over the years I have delighted in the support and positive influence of my teachers, mentors, and friends. There is no way I could fully express my gratitude—hopefully, each of you know what you mean to me—so in addition to the folks above, I would like to simply say thank you to the following folks. My extended family, especially my uncle **Linwood Griffin** and grandmother **Zelda Griffin**. From Anniston, **Connie Burleson**, **Dr. Cathy Clifton**, **Ben Cunningham**, **Theresa Haynes**, **Miranda Jones**, **Norman Morrison** and the **OCCUG**, **Carolyn Serviss**, **Glenn Spurlin**, **Tom Walker**, and the staff of **Kent's Officenter**. From Auburn, **R. Clay Bryan**, **Dr. Kai Chang**, **Dr. James Cross**, **W. Colby Gibson**, **Will Hancock**, **Seth Mason**, **Dan O'Halloran**, **Susan Reynolds**, **Dr. Marllin Simon**, and **Dr. Tom Smith** and the members of the **Auburn University Singers**. From Pittsburgh, **Dr. Natassa Ailamaki**, **Dr. Kathleen Fahey**, **Dr. Babak Falsafi**, **Brian Gold**, **Dr. Jennifer Jackson**, **Judy Jenkins**, **Bruce Klimcheck** & **Nancy Klimcheck**, **Dr. Phil Koopman**, **Dr. A. Chris Long**, **Maureen McGranaghan**, **Jen Morris**, **Dr. Bill Nace**, **Elizabeth-Jane Pavlick**, **Gerry Priano**, **Rachel Schlosser**, **Katy Shackleton-Williams**, **Dr. Charles Shelton**, **Dr. Corley Strunk**, **Dr. Reed Taylor** & **Erika Laing**, **Dr. Jackman S. Vodrey**, **Tom Wenisch** & **Shannon Wenisch**, **Roland Wunderlich**, and **Dr. Robert Page** and the members of the **Mendelssohn Choir of Pittsburgh**. From elsewhere, **Dr. Lujo Bauer**, **Dr. Randal Burns**, **Dr. Chee Yong Chan**, **Dean K. Fick**, **Dr. Cliff Martin**, **Paul Massiglia**, **Dr. Wee Teck Ng**, **Zach Peterson**, **Dr. Liddy Shriver** & **Tom Swartz**, and **Dr. Mamoru Sugie**. And, of course, **Sreevardhan** & **Vydehi Mekala** for keeping me nutritiously well-fed.

Above all, I cherish the support and encouragement I have always relied on from my parents **Cathy** & **Jack Griffin**. I am continually guided by their positive influence: their integrity and selfless generosity toward others inspires me, and their personal dedication toward the pursuit of joy and happiness challenges me to "live life large" myself. Thanks, Mom and Dad!

TABLE OF CONTENTS

List of Tables

List of Illustrations

Instead of the cross, the Albatross
About my neck was hung.

The Rime of the Ancient Mariner, II. xiv

CHAPTER 1
INTRODUCTION

This dissertation promotes the adoption of timing-accurate storage emulation as a first-class tool for the evaluation of potential computer system architectures with as-of-yet nonexistent or otherwise unavailable storage components.

1.1 Problem definition

Secondary storage is an integral and important part of most computer systems. The essence of computation is computation on data, and many data sets have long-term life spans or are simply so large as to warrant storage on an out-of-core non-volatile secondary storage system. Additionally, the cost-per-byte of secondary storage devices (such as hard disk drives) often make them a conveniently economical place for bulk data storage. However, these characteristics that make secondary storage attractive—mass data storage at relatively inexpensive cost—stem from the trade-off of low performance. (An oft-repeated axiom claims one may "choose any two of high performance, high capacity, or low cost.") When compared with the capacity of current processors to consume data, secondary storage devices fall short by many orders of magnitude in terms of the latency before data are available and the bandwidth available to transfer data. For these reasons, storage is often a limiting factor—the "bottleneck"—for many applications of computer systems. In other words, overall system performance is commonly limited by the vagaries of retrieving and storing data from the secondary storage components, as opposed to being limited by the actual computations over and transformations of that data.

Because of the critical role of storage in overall system performance, it behooves a system designer to carefully consider the options for choosing and integrating storage components into the overall system. Significant gains can be made by choosing storage configurations that maximize the most important metric of system-level goodness. For example, business-critical applications (such as web servers or inventory tracking systems) will often perform faster when they have exclusive access to a set of storage resources (disk drives, memory caches) that are not shared with other non-critical applications. However, it can be unnecessary and even detrimental to overprovision storage resources when such expenditure prevents the improvement of another part of the computer system.

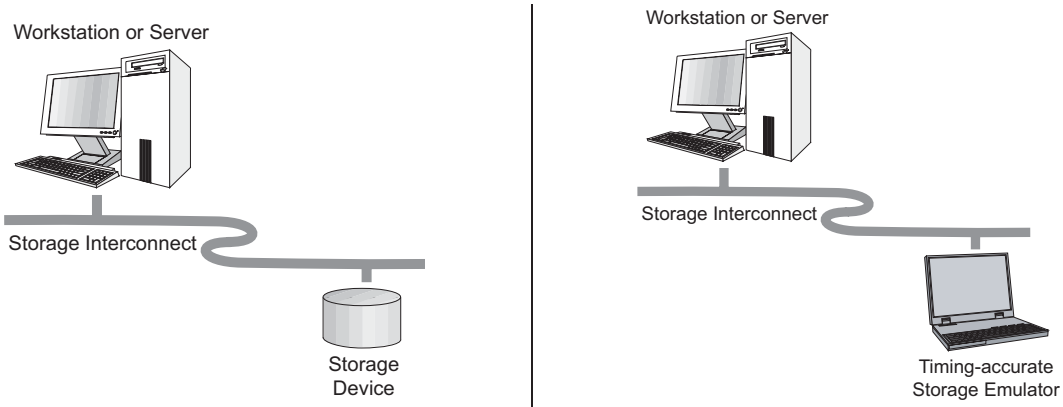
To date, a number of useful techniques have been available to assist with such decisions. At the drawing board, analytical techniques or computer simulation models can be used in conjunction with models of the expected workload to evaluate the expected consequences of a proposed device. At the laboratory bench, real system interactions can be studied whenever real devices or even storage device prototypes are available for a combined evaluation with existing and deployed hardware

and software systems. However, what these techniques lack is the realism of an actual hardware and software infrastructure combined with the use of a hypothetical or not-in-hand storage device, whose behavior or internal functionality offer an interesting potential for improving the metric of overall goodness. For example, it may be desirable to evaluate a novel scheduling algorithm for a disk drive in an existing database system, but the high cost of initially developing a prototype with that firmware functionality is prohibitive to the system designer. Such experimentation is important because complex system characteristics can hide or reduce predicted benefits of new storage components [59, 63]. Further, some new storage architectures and interfaces require both operating system modifications and new (or modified) storage components; until the new components are physically available, such collaborative advances cannot be evaluated.

One technique exists that offers an approximation of this: complete system simulation. Under this technique, the complete hardware of a computer system is simulated in enough detail to boot a real operating system and run applications. If the simulation of each component progresses according to timing-accurate models of the individual system components—such as processors, caches, buses, the primary memory system, I/O interconnects, and I/O components—this technique can be used to evaluate the performance and behavioral response of any storage device for which a simulation model exists. This technique offers the dual advantage that any or all of the individual system components can be speculative and hypothetical: in addition to looking at next year's storage device, components such as the processors and memory can be scaled up according to expected trends to simulate the overall system that will be available next year. 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. Additionally, such simulators usually run more slowly than real systems, which limits the overall scope of what variety of architectural designs can be considered.

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 1.1, 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 uses a software- or hardware-based device model to compute request service times and respond to each request after the desired service time passes, causing the performance and behavior observed by the system to match that of the device model.

Timing-accurate 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 operating systems and applications. This enables two types of experiments. First, end-to-end measurements can be made of the effects of nonexistent or unavailable storage components in existing systems. These components can be evaluated by executing one or more representative application-level workloads and evaluating both the



(a) A system with real storage

(b) A system with emulated storage

Figure 1.1: System architectures with real and 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. The examples in this dissertation use a SCSI bus as the storage interconnect.*

performance of the individual component and of the overall system. Second, end-to-end measurements can be made of the effects of non-existent storage components in modified systems. For example, it may be interesting to make changes to the storage interface to enable a heightened degree of communication between the component and an application-level or operating system-level utility. This would require modifications to both the component and the operating system device driver to support the new interface; even if the requisite changes and new functionality were available outside the storage device, experimentation is impossible without the ability to also modify the storage device. This type of modification can be difficult or even impossible given the proprietary and internally classified nature of the firmware of most disks and disk array controllers.

Perhaps more so than in the past, now is a particularly pertinent time for needing this sort of evaluation technology. The potential architectural options for non-volatile mass secondary storage are growing. For example, the gap between high-end and low-end disk drives is shifting to a new paradigm of application-specific storage—recent product development announcements are indicating a split toward consumer-grade low-cost high-capacity devices on one side and high-end highly-functional multidisk storage bricks on the other. Both product divisions are areas where the performance implications and the impact of new functionalities distributed between the operating system and the device firmware could be readily examined under timing-accurate storage emulation. Outside of disk drives, new alternative nonrotating non-volatile storage technologies such as MEMS-based storage devices are under consideration and development; until such devices are practical and widely available, timing-accurate storage emulation will become perhaps the most valuable tool for predicting their usefulness and potential impact on computer systems.

1.2 Thesis statement

Currently-adopted approaches to storage evaluation are limited in that they are unable to sufficiently mix existing prototypical or real hardware with nonexistent or unavailable storage components in order to take advantage of the best available realism when performing systems-level evaluation of component design or architectural decisions. In response to these limitations, this dissertation advocates that:

> *Timing-accurate storage emulation is feasible and enables full-system experimentation with hypothetical or unavailable storage components.*

1.3 Contributions of this dissertation

This dissertation advances four primary contributions:

1. It presents the concept of timing-accurate storage emulation and clarifies its role in the space of performance evaluation tools, including both its strengths and its limitations.
2. It demonstrates the feasibility of timing-accurate storage emulation in the context of a complete and functional implementation, and demonstrates the meshing of event-driven simulation with a real-time environment.
3. It describes a general architecture for emulation-based evaluations and mechanisms for making them highly accurate, and identifies comparable quantifications for accuracy achieved.
4. It details concrete examples of the use of timing-accurate storage emulation for real explorations of advanced storage devices and functionalities.

1.4 Overview of this dissertation

The text of this dissertation is presented in three conceptual parts corresponding roughly to the motivation, implementation, and evaluation of timing-accurate storage emulation:

- Chapter 2 discusses the role of timing-accurate storage emulation in system-level evaluations.
- Chapter 3, Chapter 4, and Chapter 5 discuss the design and implementation of a timing-accurate storage emulator.
- Chapter 6 and Chapter 7 discuss several of the evaluation opportunities enabled through the use of timing-accurate storage emulation.

Chapter 8 concludes by looking to the future of timing-accurate storage emulation. Additionally, the text is supplemented with four appendices. Appendix A and Appendix B motivate timing-accurate

storage emulation by relating our experiences with related work. Appendix C describes the low-level time interval measurement facility used in Chapter 5. Appendix D presents the full data for the experiments in Chapter 5 and Chapter 6.

1.4.1 Storage evaluations using timing-accurate storage emulation

Storage emulation is rarely used for performance evaluation of prospective storage system designs. We argue for more frequent use, noting that timing-accurate storage emulation offers a unique performance evaluation capability: the flexibility of simulation and the reality of experimental measurements. This enables experimentation on as-of-yet unavailable storage components in the context of real systems running real applications.

We highlight two classes of experiments that are possible using timing-accurate storage emulation. First, end-to-end measurements can be made of the effects of non-existent storage components in existing systems. These components can then be evaluated by executing one or more representative application-level workloads and evaluating both the performance of the individual component and of the overall system. Second, end-to-end measurements can be made of the effects of non-existent storage components in modified systems. For example, the storage interface can be extended to support autonomous data mining operations inside the storage device. Evaluating this on real devices would require modifications to both the operating system device driver and the storage device firmware to support the new interface; even if the requisite changes and new functionality were available outside the storage device, it can be difficult or even impossible to modify real devices given the proprietary and internally classified nature of the firmware of most disks and disk array controllers. However, such evaluations are feasible using timing-accurate storage emulators of these storage devices.

1.4.2 Design of a timing-accurate storage emulator

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 (or mathematical model) of that component. This allows simulated storage components to be plugged into real systems, which can then be used for complete, application-based experiments. 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.

In the timing domain, the objective of a timing-accurate storage emulator is to respond to requests externally with the exact timings associated with the device it models. In theory, this means that a host system would be unable to distinguish, using solely an analysis of request times as a comparator, whether or not the real storage component is being used when it is connected to a

timing-accurate emulator of that component. Practically, this means that the timing-accurate storage emulator must complete each storage request in real time as accurately and precisely as possible with respect to the device simulation model running inside the emulator. An important characteristic of the work presented herein is that the correctness of an emulator is independent of the correctness of the simulation model it follows. The question of executing a device model in real time, and observing and validating an emulator's behavior in comparison to the model, is divorced from the question of whether or not a particular device model has itself been validated. However, the real-world representativeness of an emulation experiment depends on the accuracy of the model.

For correct system operation, a timing-accurate storage emulator must also retain data stored on the emulated device. In most computer systems architectures, secondary storage devices are expected to provide high degrees of integrity and availability over the data that are entrusted to them. This holds true for most applications of timing-accurate storage emulation: When an emulated storage device responds to read requests, it is generally expected to transmit data that is consistent with the data that would be provided by a real device—that is, it should return data identical to what was most recently written by the host system to the specified location. Emulation software must therefore generally keep track of the data transferred during write requests in a way that enables the correct data to be returned quickly on demand; often this requires that the data for a request be somehow available in the fast memory of the emulator at the time a request arrives.

1.4.3 Evaluation directions enabled by timing-accurate storage emulation

A timing-accurate storage emulator takes a versatile role in storage system evaluations. One use of timing-accurate storage emulation is as an inexpensive and readily-available alternative to purchasing and installing existing storage products when evaluating multiple potential architectures. A second use of timing-accurate storage emulation is to evaluate the potential of introducing standalone hypothetical storage components into computer systems. These devices could represent either evolutionary changes to existing products or revolutionary storage designs. A third use of timing-accurate storage emulation involves evaluating external system architectures in which host software explicitly interacts with new or modified functionalities in the emulated storage subsystem. As needed, the emulation environment can additionally support extended interfaces between the host system (or another external entity) and the emulated storage components; these interfaces can then be used to transfer meta-information or functional instructions in either direction to support the new functionality.

We built a functional timing-accurate storage emulator and demonstrated its use in experiments represented by the second and third categories above. To explore standalone hypothetical storage components in computer systems, we configured our emulator with three device models representing a currently-available production disk drive, a hypothetical 50,000 RPM disk drive, and a hypothetical MEMS-based storage device, and executed three distinct application-level workloads against these three emulator configurations. To explore new system architectures with expanded de-

vice functionality, we applied the principles of timing-accurate storage emulation in an investigation into storage-based intrusion detection systems. This experimentation demonstrates the feasibility of including intrusion detection capabilities into a standalone processing-enhanced disk drive, and also demonstrates how existing communications paths may be used by an operating system to transmit and receive information regarding the configuration and operational status of such an intrusion detection-enhanced device. The successful results of this experimentation, coupled with previously published work on active, intelligent, and semantically-smart disk systems, again heralds the call for the inclusion of processing capabilities inside peripherally-attached computer system components.

1.5 Summary

Timing-accurate storage emulation offers the opportunity to investigate novel uses of storage in computer systems, permitting forays into the space of hypothetical device functionalities without the difficulties of developing and supporting extensively nonstandard or novel interface actions in prototype or production systems. This dissertation demonstrates that there is a current and pressing need for a new storage evaluation technique, and that it is feasible to design and construct a timing-accurate storage emulator and to use an emulator for interesting systems-level experimentation.

CHAPTER 2
BACKGROUND AND MOTIVATION

Despite decades of practice, performance evaluation of proposed storage subsystems is often 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 I/O-related disciplines such as research into networking or file systems.

This chapter presents a case for widespread use of timing-accurate storage emulation as a tool for analysis and evaluation of the use of not-yet-existing storage devices in real computer systems, arguing that the technique overcomes certain limitations of other approaches. It summarizes the role of timing-accurate storage emulation in the context of existing storage performance evaluation techniques (Section 2.1), discusses the strengths and weaknesses of timing-accurate emulation as a storage evaluation technique (Section 2.2, page 17), and discusses related work for timing-accurate component emulation (Section 2.3, page 20).

2.1 Evaluating the impact of storage components in computer systems

Storage emulation is rarely used for performance evaluation of prospective storage system designs. This chapter 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. 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. However, these evaluation techniques are limited in terms of both their inability to execute real application-level workloads and their inability to fully account for system-level interactions with the storage device under test. Further, some new storage architectures and interfaces require both OS modifications and new (or modified) storage components—until the new components are available, only timing-accurate storage emulation allows such collaborative advances to be tested and their performance evaluated in the context of real systems.

This section discusses the currently-available techniques for evaluating hypothetical storage devices, the available techniques for storage experimentation in the context of complete systems, and the role of timing-accurate storage emulation in relation to these techniques.

2.1.1 Techniques for evaluating hypothetical storage devices

Computer system implementors often must consider a wide range of variables when designing application- or role-specific computing solutions, including often application-behavior-dependent tradeoffs among such factors as complexity, cost, deployment effort, maintenance sensitivity, performance, and upgrade flexibility. Because secondary storage can significantly impact each of these variables, the system builder must be especially careful to understand the implications of specific design choices for the storage subsystem. For example, choosing a simple redundancy-based data protection scheme for the disk array supporting a frequently-updated transactional database may enable a faster initial deployment but may also cause an unsatisfactorily degraded level of performance during the reconstruction phase following the failure of an individual disk.

By the same token, the researchers and developers who continue to drive innovation in storage systems components, both in the corporate and academic arenas, are interested in quantifying the true nature of the system-level impact of a proposed hardware or software modification. Unfortunately, implementing every drawing-board idea can prove prohibitively costly in terms of both temporal and physical resources, and depending on the timeline of the expected deployment of the product (or the computer system above) might well be impossible.

In each of these environments, there is a desire to evaluate alternative storage subsystem designs or architectures with postulated but as-of-yet unrealized components. Once an abstract device model is available for the unrealized device—that is, once the modeler develops a conceptual understanding of how the novel device will behave—several classes of evaluation techniques exist that permit exploration of such a device. These include analytic modeling, simulation-based device modeling, and simulation-based full-system modeling, the latter of which is discussed in the following subsection. Each class of techniques provides different levels of fidelity in terms of the accuracy, completeness, and representativeness of the model and evaluation results as compared to the ultimately deployed storage system. These levels of fidelity are generally tunable in terms of the model development time and validation effort required for the initial construction of the device model.

Analytic models for storage use stochastic and queuing theory to create probabilistic models that quantify a device's performance response to input workloads. Analytic models have been used for decades in investigations into individual device behaviors [16, 153]. More recently, attention has turned to the application of analytic models to understanding the behavior of arrays of devices [167, 169, 180]. (See selected references from Shriver, Hillyer, and Silberschatz for an extensive history of the development and use of analytic models in studies of disk drives, disk arrays, and tape systems [154, §3.2].) Once a baseline model is created for a device, it can be replicated and extended to account for multiple alternate device configurations. Synthetic workload models based on characterizations of the expected workload are used to drive analytic storage models at validation and experimentation time. Evaluations using analytic models are advantageous in that they permit rapid development of an experimental environment, especially when detailed internal

characteristics of the storage device (such as firmware algorithms or optimizations) are not fully understood. Additionally, analytic models are generally conceptually easier to understand than the alternatives of simulation models or empirical models. However, due to the probabilistic nature of such analyses, experiments using analytic models of storage are best directed toward quantifying the statistics of aggregate device behavior under a workload rather than understanding individual, time-critical device responses to specific requests.

Simulation-based models for storage combine a detailed description of a device's state-based mechanical and algorithmic behavior with knowledge of how some or all previous storage requests contributed to the current device state; this allows an experimenter to quantify a device's performance response to input workloads. Development and use of high-accuracy simulation-based models has progressed rapidly over the years, which has enabled detailed investigations into the design, behavior, and performance of disk drives [47, 63, 70, 90, 103, 111, 149, 181], disk arrays [6, 40, 77, 91, 176] hierarchical or distributed memory caches [11, 177], tape systems [81, 80], and new storage technologies (see Schlosser [144] and selected references therein). Simulation-based models are generally more complex to build than analytic models, as each functional component (both mechanical and algorithmic) must be individually considered regarding its contribution to the model's accurate portrayal of a real device's responses to internal and external stimuli. Additionally, greater care and effort must be taken at evaluation time to ensure that the focus remains on quantifying true overall system effects instead of potentially unimportant trends at the device's simulated internal components. However, this ability to concentrate on minute details is an attractive feature of simulation-based modeling, in that it enables fine-grained observation of how individual physical design decisions or modifications affect the externally-observed overall behavior of the device. Additionally, simulation-based techniques enable fine-grained analysis of a device model's discrete responses to individual storage requests, as well as considerations of how the device behavior and state is affected by a group of specific temporally-ordered storage requests.

2.1.2 Experimentation in a full systems context

Unfortunately, there are several factors that limit the desirability of analytic modeling and simulation-based modeling. In particular, these techniques limit consideration of real application workloads and complex real-system effects. Input workloads to these techniques are generally either (possibly temporally scaled) replayed traces of a previously-measured request stream, or an artificially-generated sequence of requests based on a statistical sampling of previously-measured request streams. Evaluations using trace replay or synthetic workload generation are susceptible to errors introduced by the mismatch between the configuration of the initial trace-gathering system and both the experimental device's internal characteristics and the real system's feedback effects. Request interarrival times for a workload are not necessarily independent of the device's response time for individual requests. When the characteristics of the trace-gathering device do not match those of the experimental device—for example, if the devices differ in their physical capacity, geometric layout of

data, firmware algorithms, or even in the interconnection buses used to join the device to the external system—this can cause a loss of realism in the representativeness of the experimental results. Similarly, errors will likely be introduced when the hardware and software of the trace-gathering system—for example, the selection of processor or cache components, or the choice of operating system—differs from the configuration of the real, to-be-deployed system. Worse, trace-based evaluation may not adequately capture the feedback between the storage device and the the external system that impacts overall (i.e., non-I/O-specific) metrics of architectural or experimental goodness, as discussed by Ganger and Patt [63].

Techniques involving experimentation with real applications executing on real external hardware provides the most desirable level of realism. These techniques are especially appropriate when the desired storage component is physically available for experimentation; i.e., when any hardware modifications or software implementations have already been fully realized in a prototype or production device by the device engineers. For example, this technique is used by industrial storage device developers to look at cutting-edge performance enhancements: to evaluate the gains of including a faster spindle motor in a disk, a faster motor may be built and placed inside an otherwise minimally modified production disk (known as a "mule") [130]. Unfortunately, although it is beneficial to evaluate real devices in experimental environments, consideration of a large number of prospective devices may dramatically increase the overall cost and effort of such experimentation. Additionally, when considering hypothetical devices, it is seldom possible to quickly and inexpensively prototype major refinements to existing devices. This is especially difficult for researchers not affiliated with the device's manufacturer who therefore do not enjoy ready access to proprietary firmware and manufacturing facilities.

A technique that contains the realism of external system interactions combined with evaluations of hypothetical storage devices is full-system simulation. Examples of timing-accurate full-system simulators are the Virtual Machine Emulator [25], SimOS [79], Simics [113], and Mambo [18]. The SimOS and Simics simulators in particular have enjoyed widespread dissemination and use in computer system evaluations. Under the technique of full-system simulation, 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., processors, caches, buses, memory system, I/O interconnects, I/O components), it can be used for performance evaluation. Further, by manipulating simulator parameters, the effects of new storage devices on hypothetical machines (e.g., with 10 GHz CPUs) can be evaluated [132, 146]. 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. Real computer systems are complex and they continue to advance rapidly. For example, the SimOS machine simulator required several years of effort to create and validate; just a few years later, its hardware models were out of date, the CPU instruction set it emulates was phased out, and source code for the OS that it boots became difficult to acquire. In addition, full-system simulators usually run much slower than

real systems, which increases evaluation time and makes it difficult to evaluate interactions between computer systems and external entities such as workstation users or network-attached components.

2.1.3 The role of timing-accurate storage emulation

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 (or mathematical model) of that component. This allows simulated storage components to be plugged into real systems, which can then be used for complete, application-based experiments. 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.

Table 2.1 illustrates a spectrum of seven techniques for evaluating storage designs, placing each of the techniques from above into a common context. Techniques toward the top 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 toward the bottom generally produce more believable results: more accurate, more inclusive of complex system effects, and more representative of the effects under real workloads. Each of the seven techniques shown are 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 viewed with skepticism, 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 [57], I/O traces ignore system feedback effects [63], and available traces are often out-of-date—in fact, many storage researchers still rely on the decade-old HP traces [135] from 1992. 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.

Timing-accurate 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. These components can then be evaluated by executing one or more

	Initial device development effort	Subsequent device modification effort	Ability to evaluate hypothetical devices	Inclusion of end-to-end external interactions	Experimentation using real external hardware	Ability to explore extended interfaces to storage	Experimentation with hypothetical computer systems	Experimental execution time	Real-world representativeness of results
"Back-of-the-envelope" calculations	Low	Low	Y				Y	Fast	—
Analytic model-based calculations	Medium	Low	Y				Y	Fast	Low
Storage subsystem simulations	Medium	Medium	Y				Y	Fast	Medium
Full-system simulations	High	Medium	Y	Y		Y	Y	Slow	Med/High
Timing-accurate emulation experiments	Medium	Medium	Y	Y	Y	Y		Normal	High
Physical prototype experiments	High	High		Y	Y	Y		Normal	Highest
Production device experiments	High	—		Y	Y			Normal	Highest

Table 2.1: Storage performance evaluation techniques. *This table compares techniques across a variety of metrics. Timing-accurate storage emulation provides the unique ability to explore nonexistent storage components in the context of full systems executing real applications. A "Y" designation indicates the technique enjoys the advantage described by the column header. The "Med/High" designation for full-system simulation indicates the difficulty of maintaining accurate system models given the rapid advancement of such technologies.*

representative application-level workloads and evaluating both the performance of the individual component and of the overall system. Second, end-to-end measurements can be made of the effects of non-existent storage components in modified systems. For example, the storage interface can be extended to support autonomous data mining operations inside the storage device. Evaluating this on real devices would require modifications to both the operating system device driver and the storage device firmware to support the new interface; even if the requisite changes and new functionality were available outside the storage device, it can be difficult or even impossible to modify real devices given the proprietary and internally classified nature of the firmware of most disks and disk array controllers. However, such evaluations are feasible using timing-accurate storage emulators of these storage devices.

2.1.4 Comparison with full-system simulation

Of the existing storage evaluation techniques, timing-accurate storage emulation is most similar to full-system simulation in terms of the real-world representativeness of experimental results and the types of enabled storage system evaluations. Full-system simulation shares many of the advantages of emulation, in that both techniques can be used to evaluate hypothetical devices in the context of their interactions with complete computer systems. Identical software-based storage models can be used in both environments, eliminating any difference in storage model development and validation time between the approaches. However, few projects have used full-system simulation specifically for evaluating the interaction between computer systems and storage; examples of these include the work by Schlosser et al. investigating the architectural implications of integrating MEMS-based storage technology into the memory hierarchy [146], and the work by Gurumurthi et al. for profiling power dissipation in a power-optimized computer system [76]. We postulate that this is not due to the inapplicability of full-system simulation for storage evaluations, but rather the historical emphasis of full-system simulation on exploring components further up the memory hierarchy—as demonstrated by Maynard, inattention to disk-level validation for the default disk module in full-system simulators can cause performance misprediction [115]—as well as the propensity of storage system researchers to press forward with building real systems whenever application-level experimentation is desired. Regardless, we note that successful storage evaluations often do not need many of the specific advantages provided by full-system simulation, and conversely that there are advantages to using timing-accurate storage emulation that full-system simulation does not share, as discussed in the remainder of this section.

Many projects have used full-system simulation to good effect in computer system component evaluations. The technique offers many experimental-time advantages, which have doubtless contributed to the wide adoption and use of the full-system simulation: Full-system simulators can provide detailed summaries of performance and behavioral interactions of individual system components. The simulation environment has full control over clock advancement, making it straightforward to temporarily halt progress and examine intermediate system state during experimentation

and additionally introduce anomalies wherever desired for debugging purposes. Importantly, evaluations using full-system simulation (as well as those using several of the other non-emulation techniques) can be configured to simulate hypothetical computer systems as well as hypothetical storage components. This can often be done by simply changing the simulation parameters in a configuration file, or alternatively the simulation software can also be manually tuned to reflect the characteristics of the desired system. The most widely used simulator today is Simics, which has a company with a business model behind its development, which suggests the likelihood of continued retail availability of full-system simulators.

Unfortunately, full-system simulators have yet to provide accurate operation of experimentally interesting modern computer systems at real-time speeds. Further, full-system simulators require prodigious amounts of effort to develop and maintain: as an example, the Simics developers claim to have spent 50 person-years of development generating approximately one million lines of software to support the simulator in its various configurations [113]. For comparison, our timing-accurate storage emulator required approximately two person-years of development. These limitations highlight three key advantages of timing-accurate storage emulation over full-system simulation:

1. Timing-accurate storage emulation can proceed in real-time, which has the advantage of decreasing overall evaluation time, as well as permitting the inclusion of real interactions in real-time between the real computer system (with the emulated storage) and people using the system or other computer systems.

2. Timing-accurate storage emulation provides the best possible realism for experimentation with hypothetical storage devices by incorporating as many real components as are possible into the experimental environment.

3. Timing-accurate storage emulation requires much less effort to develop and deploy an accurate experimental infrastructure.

Additionally, timing-accurate storage emulation gives the evaluator the ability to concentrate on the development, validation, and instrumentation of the desired storage component model, without needing to worry about correctly configuring the simulation environment or interpreting the validity of a full-system simulator's execution.[2.1]

[2.1] Of course, when using timing-accurate storage emulation the evaluator must instead worry about correctly configuring the emulation environment and interpreting the validity of the real-time emulator. Anecdotally, we found it straightforward to quickly integrate local emulation capabilities into an operating system, the ease of which is mirrored by the success other projects using local emulation as enumerated in Section 2.3.2. (The concepts of local and remote emulation are introduced in Section 3.2.2.) We found it substantially more difficult to develop a stable platform for remote emulator development—this required approximately one graduate-student-year of effort—due mostly to the limited availability of documentation describing the complex interactions among the emulation software, the operating system software that supports emulation, and the target-mode bus adapter firmware that supports emulation. As remote emulation functionality continues to grow in popularity and more documented software utilities become publicly available, we anticipate remote emulator development to become more straightforward. In either case, we expect the configuration and validation techniques for timing-accurate storage emulators described in Chapter 5 will also simplify the evaluator's task.

2.1.5 Previous experiences with non-emulation-based storage evaluations

Two case studies from our recent experience with non-emulation-based storage evaluation are presented in Appendix A and Appendix B. These case studies identify several experimental situations where the use of timing-accurate storage emulation would have been applicable and desirable. The projects described in the case studies used similar to timing-accurate storage emulation in terms of development effort and representativeness of results: full-system simulation (of MEMS-based storage devices, referred to here as Project A) and production-device experimentation (with track-aligned extents, Project B). These case studies highlight the utility of timing-accurate storage emulation in two ways. First, the project descriptions show several types of experiments where timing-accurate storage emulation would be relevant and would capture many of the same results. Second, the project evaluations suggest opportunities for continued evaluations using new devices or new workloads; timing-accurate storage emulation would have allowed pursuit of these new directions, whereas the existing techniques could not support such investigations.

Projects A and B establish well the utility of existing storage evaluation techniques. Project A demonstrates system performance and resource consumption implications that arise when introducing hypothetical non-volatile storage devices as complements or replacements for disk drives in the computer system memory hierarchy. Project B demonstrates how an operating system can use device-specific knowledge in its data allocation and access decisions, with the result of increasing the efficiency of requests at the storage device and therefore reducing application-level stall times. Unfortunately, each project also demonstrates the limitations of scope and representativeness of the results available using each technique. Using full-system simulation in Project A, we had no choice but to use the single available operating system and hardware platform supported by the simulation environment; both of which were years out of date and were only questionably representative of the environments into which the hypothetical device would be introduced. Additionally, we were unable to run experiments involving user interactions with the system containing the hypothetical device. Using production-device experimentation in Project B, we were unable to evaluate any alternative products that lacked the particular firmware support (zero-latency access) that our efficiency improvements relied on. Were were also unable to adjust or tamper with any hardware characteristics or firmware functionality to determine where other opportunities might exist to increase the efficiency of operating system-level storage interactions.

Our experiences with these two projects indirectly emphasize the power and potential of timing-accurate storage emulation. First, all of the experimental results recorded during our experimentation would have been equally obtainable using the technique of timing-accurate storage emulation, except with a higher confidence in the representativeness of the full-system simulation results due to the use of applications running on a real operating system and real hardware. Second, pursuit of several new ideas identified during experimentation (other opportunities to improve efficiency, or user interactions with hypothetical systems) were difficult or impossible to pursue using the existing techniques but would have been possible using timing-accurate storage emulation.

2.2 System-level experimentation using timing-accurate storage emulation

A timing-accurate storage emulator can take a versatile role in storage system evaluations. An emulator may be used as an inexpensive and readily-available alternative to acquiring demonstration or example hardware when choosing which currently existing storage products to purchase. Or, it can be used to evaluate the impact of storage devices representing significant futuristic technological progress, to get an early feel for which innovations would return the most beneficial return if pursued. The scope of which hardware and software components are emulated can be contained within a single device or controller, or can span an entire multi-device hierarchical storage subsystem. This section discusses these roles, as well as the limitations on effectiveness and applicability that are inherent to timing-accurate storage emulation.

2.2.1 Approaches benefitting from emulation

There are three experimental domains for which timing-accurate storage emulation is especially pertinent: those looking at deployment options involving currently-existing storage products, those evaluating the potential of nonexistent storage components, and those evaluating modified external system architectures or interactions with modified or hypothetical storage device functionality.

One use of timing-accurate storage emulation is as an inexpensive and readily-available alternative to purchasing and installing existing storage products when evaluating multiple potential architectures. This of course would rely on the availability of storage devices models, whether provided by the device manufacturer or an independent testing and development entity. The cost and complexity of developing or deploying an emulation product would be amortized across the savings of time and expenditure of obtaining product samples for evaluating, assuming such samples are even available.

A second use of timing-accurate storage emulation is to evaluate the potential of introducing standalone hypothetical storage components into computer systems. These devices could represent either evolutionary changes to existing products or revolutionary storage designs. This would be appropriate in an advanced host-system product development environment, where one goal is to evaluate high-end external system prototypes with the storage facilities that will ultimately be available at deployment time. Alternatively, this would be appropriate in a storage product research and development environment, where one goal is to evaluate the overall performance impact of potential device modifications in the physical hardware or even functionality of the component controller or an individual device's firmware. The case studies involving MEMS-based storage devices and track-aligned extents, presented in Appendix A and Appendix B respectively, are examples of experiments that would fall into these categories if performed using timing-accurate storage emulation.

A third use of timing-accurate storage emulation involves evaluating system architectures in which external components explicitly interact with new or modified functionalities in the emulated storage subsystem. As needed, the emulation environment can (explicitly or implicitly) support

extended interfaces between the external components and the emulated storage components; these interfaces can then be used to transfer meta-information or functional instructions in either direction to support the new functionality. Such functionality can exist solely within the domain of the emulated storage system, which would allow evaluations with unmodified host systems. Alternatively, the approach may be used to evaluate new system-level functionalities or performance improvements based on enhancements split between both the emulated components and the host operating system or evaluation applications. Our investigation into storage-based intrusion detection systems, described in Chapter 7, is an example of this type of experiment using the principles of timing-accurate storage emulation.

2.2.2 Scope of emulated components

We define the *scope* of a timing-accurate storage emulator as the statement of which components' timings and behavior are determined wholly by the emulation software. The scope of an emulator may include just a single device, such as an individual disk drive, or may involve many system components. For example, the emulator's internal model can include bus interconnection components, multiple storage devices interacting together, device array controllers, or even a hierarchy of components such as a non-volatile cache in front of a disk. This is a design-time tradeoff that depends on several factors. One dependency involves what hardware support is available for emulation; as discussed later, some forms of storage emulation require both special hardware functionality in the bus adapter and special operating system support for target-mode operation. Another dependency involves what real external hardware and software components are available for building the experimental system. If the evaluation system will contain a futuristic storage interconnect, such as a bus with specific device arbitration characteristics, it would be appropriate to include the interconnect within the scope of the emulated subsystem. A third dependency involves which component models are available for emulation, and whether such models have been validated against real components.

In terms of the input workload used to drive the performance and behavior of a timing-accurate storage emulator, any of the workloads used for other advanced storage evaluation techniques are appropriate. This ranges from artificially-generated synthetic workloads; unscaled or scaled traces of storage requests taken from past experimentation; system-level microbenchmarks [133]; and application benchmarks such as the PostMark benchmark [96], the SSH-build benchmark [151], and the TPC database benchmarks from the Transaction Processing Performance Council. More importantly, it includes real application workloads; much of the motivation for timing-accurate storage emulation stems from its ability to execute real applications on both modified and unmodified external systems. Further, as discussed above, the emulator can use more workload information than is normally available during experimentation; the emulation environment can support meta-information passing across the storage interconnect to allow the sharing of application-level or operating system-level information with the storage components, enabling more "intelligent" operations inside the storage subsystem.

2.2.3 Limitations of timing-accurate storage emulation

In spite of its strengths and advantages, the technique of timing-accurate storage emulation is not a panacea and its use is not appropriate in all evaluation scenarios. An evaluator must be cognizant of several limitations to the approach that can dilute its effectiveness or applicability under certain circumstances.

One limitation to timing-accurate storage emulation is that the choice of external system components are limited to those that are physically available to the evaluator. For example, one can only use timing-accurate storage emulation to evaluate a database system with novel storage subsystem functionality if the non-storage portions of the hardware and software composing the database system are on hand. Similarly, when evaluating an architecture with the predicted storage subsystem performance of next year's disk products, such an evaluation can only be made in the context of current hardware system products or prototype platforms. One observer has wryly noted that "timing-accurate storage emulation provides the ability to evaluate tomorrow's storage in today's computer systems." This is fair criticism, and the evaluator must weigh the pros and cons of this technique over alternate methods such as full-system simulation, where it is possible to simulate applications running on any futuristic hardware but where it can be extremely difficult to build and maintain accurate models of the same. On the other hand, the technique of timing-accurate storage emulation does enjoy the realism of using existing systems. Use of this technique can be construed as a valuable incremental step towards an ultimate evaluation using a real storage prototype, making use of the best resources available until such prototype evaluation is possible.

Another limitation concerns the scarcity of validated physical device models for existing storage products and speculative device models for future devices, and especially the question of whether the results of emulation-based experiments are truly representative of the results that would come from a real device. Whether using simulation models or analytic models as the basis of the emulator's performance and behavior, emulation-based experiments share the shortcomings of all model-based experimentation in that the accuracy of the emulator can be no greater (and is potentially less) than the real-world representativeness of the model. Fortunately, the facts that highly-accurate validated models exist for disk drive products [141], and that there continues to be interest in building simulation models and analytic models for disk array products [169] and other secondary storage components [73], suggest that models will continue to be available to evaluators. If timing-accurate storage emulation develops into a well-used tool, storage product manufacturers might find it practical and economically feasible to release validated models of their devices to support external emulation-based evaluation. Regardless, as implied elsewhere and discussed in Section 3.2.4, this work assumes the existence of an acceptable device model and focuses on the techniques necessary for replicating the model's exact behavior in an emulated environment.

A third limitation involves the peak performance of the emulation system itself. As discussed later, there is in many cases only limited hardware and software support for certain emulation functions, such as the ability for a computer system to receive requests and send replies on a storage

interconnect (as opposed to the usual role of sending requests and receiving replies). Further, for some desirable system designs—for example, our initial expectation to use the Linux operating system as the foundation of our emulation software—the requisite support is currently nonexistent in the mainstream distributions and it is nontrivial to implement [121]. Also, limited support for emulation functionality in the interconnect adapter hardware can result in potentially severe limitations on the minimum request time, maximum data rate, and maximum request interarrival rate. For example, the implementation used in this dissertation has a minimum request time of approximately 0.1–0.3 ms per request depending on the bus adapter used, as discussed in Section 5.4. Much smaller minimum times and maximum rates can be achieved by colocating the emulation software on the experimental host system, but this increases the risk that the emulation software will have a performance impact on applications running on the now-modified host system.

2.3 Survey of related work

Much previous work related to timing-accurate storage emulation is presented earlier in this chapter: Section 2.1 discusses non-emulation-based storage evaluation techniques, refers to projects that make use of such techniques, and compares the techniques with timing-accurate storage emulation. This section notes additional related work in three areas: it surveys published models that capture the behavior and performance of storage devices; it identifies projects that have used emulation-like approaches for storage component evaluation; and it discusses experimental approaches that are taken in the related I/O-centric disciplines of networking and file systems.

2.3.1 Storage subsystem modeling

Much work has gone into the development of efficient and representative storage device models. We classify work on model development into the three categories of empirical models, analytic models, and simulation models. The reader seeking a comprehensive introduction to issues surrounding the methodology for creating and validating performance models in each of these categories is referred to the treatment by Jain [94]. This section discusses recent work toward model development in each category.

Several projects have developed empirical methods that abstract a disk's mechanical characteristics into a lookup table of expected disk access times [8, 126, 163], which can then be used in predicting individual request service times during experimentation. Such table-based models have the advantages of fast and efficient operation (in terms of the on-line processing requirements), abstracted and automated off-line or on-line table creation, and some degree of real-world representativeness in relation to the mechanical operation of a device's hardware components. As to the disadvantages of the approach, an actual device may be needed at table creation time, and it can take a large number of storage requests to populate the entries in the table. In order to maintain reasonable table sizes, the table may need to abstract out per-request algorithmic effects such as

scheduling and caching policies. Additionally, it is difficult to update the table to reflect device modifications due to the abstract nature of the device model. However, for emulation of very high-performance secondary storage devices, table-based approaches may allow experimentation when other approaches fail due to their inability to complete detailed per-request simulation before the request deadline.

Models developed under these approaches have generally proceeded along two tracks, emphasizing either single-device systems or multi-device systems such as disk arrays. Recent analytic models are presented by Shriver, Merchant, and Wilkes for individual disk drives [153] and by Varki et al. for disk arrays [169]; previous work in storage-based analytic models includes a series of incremental and revolutionary models capturing ever-finer-grained detail of the performance and behavior for disks, disk arrays, and tape systems, as discussed in the related work of these references and as surveyed by Shriver, Hillyer, and Silberschatz [154, §3.2].

Ruemmler and Wilkes published a brief tutorial on simulation-based modeling of individual disk drives [136]. A disk simulator created by Kotz, Toh, and Radhakrishnan [105] was later used as a core disk module in the SimOS full-system simulator [79]. Ganger, Worthington, and Patt developed the general-purpose DiskSim storage simulation environment [64] which led to the development of a disk characterization and model creation utility [141] and a publicly-available database of validated simulation models for various disk drive products [56]. Work directed toward disk array simulation includes the Pantheon simulator [174], the raidSim simulator [37, 101], and the RAIDframe framework [41].

Outside the realm of disk-based storage, simulation models of hypothetical MEMS-based storage devices were developed by Griffin et al. [73] and Madhyastha and Yang [112]. Various analytic models and simulation models for tape systems are presented by Drapeau and Katz [49], Golubchik, Muntz, and Watson [71], and Hillyer and Silberschatz [82].

Our implementation of a timing-accurate storage emulator[2.2] makes use of simulation models for single disk devices and MEMS-based storage devices that were built for the DiskSim simulation environment. Additionally, many of the other models and environments would be appropriate for evaluation approaches using timing-accurate storage emulation; as discussed in Section 5.1.1, a primary limitation on which models are available for emulation purposes is that the amount of processing time required to execute the model and compute the completion time of any given request (excluding the inactive time during which request is queued at the emulator) must be less than the actual service time for that request.

2.3.2 Storage emulation evaluations

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 to look like

[2.2] Details of this implementation and its interactions with the simulation models are presented in Chapter 5.

traditional NFS file servers [83]. Computers from Apple Computer, Inc., support file transfer operations using native target-mode operation: When two Macintosh-brand computers are connected using a FireWire interconnect, one appears as a locally-attached disk on the other. In addition, we have been told anecdotal stories of the use of emulation in industry for development and correctness testing of new product designs; support for the requisite target-mode operation is available and marketed for this purpose in several of the hardware products and operating systems discussed in Section 4.3. A less direct example is the practice of evaluating non-volatile RAM by simply pretending that normal RAM is non-volatile [35, 62].

Several projects have used storage emulation to build and evaluate systems containing new storage functionalities. Generally speaking, these projects introduce new software that intercepts storage requests destined for a real disk and emulates the desired high-level functionality before allowing the request to proceed to the real disk. This software can either run locally on the system under test or remotely on special hardware dedicated to the emulation task.[2.3] Although disk-accurate request timings are not (to our knowledge) a stated design goal of such projects, the projects use real disks to determine request times in a similar manner to the timing management for our evaluations of a storage-based intrusion detection system as described in Chapter 7. An example of local software interposition is the implementation of the RAIDframe framework [39]. RAIDframe is intended for use in researching, verifying, testing, and producing RAID systems, and could be used both as a storage subsystem simulator or as an application-level or OS-level RAID controller for evaluating real applications running against an emulated disk array. Examples of remote software interposition include the StarFish project at Bell Laboratories [54] and the Network Attached Secure Disk (NASD) project [68]. The StarFish project used emulation to evaluate the use of host-transparent, geographically-replicated block storage devices to increase the data availability and reliability metrics provided by the storage subsystem. The hardware and software components used for target-mode emulation support in StarFish are very similar to experimental setups used for our experiments and described in Chapter 4. Some of the investigations in the NASD project used emulation to evaluate the performance, computational requirements, and scalability of applications running against storage devices with novel object-based interfaces. As with our investigations into timing-accurate storage emulation, the NASD evaluations uncovered implementation-specific limitations on evaluation-time performance that did not necessarily reflect the true performance limits of a real device.

We are aware of only a few previous cases of timing-accurate storage emulation being used for performance evaluation. One example is the evaluation of the Galley Parallel File System by Nieuwejaar and Kotz [118]. Galley is composed of computational processing nodes and I/O processing nodes that together support the operation of high-performance parallel scientific computing applications. To avoid false inflation of the observed disk performance due to file system prefetching and caching effects on the I/O nodes, the evaluators integrated a thread running a disk simulation

[2.3] This distinction parallels our discussion of local and remote emulation in Section 3.2.2.

model into each I/O node. This thread determines the actual completion time of a request: when a request arrives, the simulation model calculates the time required to complete the request, then the thread suspends itself for that length of time. This thread is similar in functionality to the timing manager in our emulator design.[2.4] Galley additionally supported a limited data management functionality, in that the disk simulation thread maintained a small cache to store meta-data blocks important to the file system operation. Unlike the data manager in our implementation, the contents of ordinary data blocks were not preserved by the thread cache, presumably due to the limited amount of RAM available to the thread on the emulation system.

Another previous example of timing-accurate storage emulation is the evaluation by Wang, Patterson, and Anderson of the virtual-log logging strategy for Programmable Disks that support eager writing [171]. 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 eager writing, the experimenters used local emulation to interpose support for virtual logging in the firmware of an emulated disk. Timing management for emulated disk requests is provided in a similar manner to that of Galley, in that the kernel thread sleeps for the length of time determined by the disk simulator. The design is additionally augmented with a small RAM cache that stores and returns written file data.

A more recent example is the exploration of applications of distributed computing on active storage devices by Wickremesinghe, Chase, and Vitter [173]. This evaluation used local timing-accurate emulation of both storage and network components to demonstrate the potential of dynamic adaptation on active devices. The disk and network simulation models used in the emulation environment were simple—for example, the disk simulator did not model detailed seek and rotational times—but these simple models were appropriate for the particular experimentation due to the known fully-sequential nature of the I/O workload. Evaluations such as this that use workloads with well-known or straightforward characteristics are good candidates for implementing the multiple-fidelity model support in limited-processing emulation environments discussed in Section 5.1.1.

The designs of the timing-accurate emulation components in Galley and the Programmable Disk are similar to our design as presented in Chapter 3. However, there are some notable differences between this work and previously-published research: We present a technique for synchronously executing an event-driven simulation model in real time,[2.5] enabling our simulation model to react to additional external events (such as new request arrivals) that occur during the execution of a request. We develop a method for quantifying and mitigating the per-request errors introduced by the emulation infrastructure. Additionally, whereas the previous investigations used simulation models of existing disk products, we use a general-purpose disk simulation environment to evaluate both current and future storage component designs.

[2.4] The functions of timing and data management for timing-accurate storage emulators are described in Section 3.3.

[2.5] As discussed in Section 5.3.1, our implementation introduces the run-synchronously approach, whereas it appears that the Galley and the Programmable Disk use the run-to-completion approach.

2.3.3 Evaluations in related I/O-centric disciplines

The importance of capturing real system effects through experimentation with complete computer systems, timing-accurate emulation, or full-system simulation is well established in other areas of computer system evaluation, including experiments focusing on network and file system research and development.

Evaluations of network components and protocols are generally system-level in that they concentrate on developing the context of a component modification or protocol design's effect on a complete, end-to-end network architecture. To accomplish this, actual experimental computer networks are often built and evaluated in the local area or the wide area. However, when scale considerations do not permit construction and deployment of substantial networks, full-system simulation [22] or timing-accurate network emulation [5, 50, 119, 168, 172] techniques are employed. Timing-accurate network emulation parallels our description of timing-accurate storage emulation: real hosts interconnected by an network emulator observe normal packet send/receive semantics and performance that accurately reflects a simulation model. Some controllable performance effects of the emulated network include propagation delays, bandwidths, and packet losses. The use of these full-network evaluation techniques enable real system benchmarking and design analysis whose representativeness would not otherwise be fully captured by component-only analysis.

Evaluations of file system designs, investigations into the operational behavior of file systems, and quantifications of externally-observed file system performance are almost always performed in the context of file system prototype experimentation or production file system experimentation in a complete systems context. This is due in part to the availability of stable and extensible operating systems upon which to integrate file system modifications or build new file system designs, and simplified implementation due to semi-rigid specifications regarding the interface between a file system, the users of the file system, and the remainder of the operating system. In addition to the veritable cornucopia of file system projects that have used prototype-based or production-based experimentation, however, there has also been interest in using file system simulation (combined with varying degrees of emulator-like integration into real-world environments) as a tool for rapid prototyping of file system modifications with a comfortable degree of real-world representativeness of the results [13, 20, 104]. Thekkath, Wilkes, and Lazowska report that carefully-designed file system simulation is able to capture many of the complex interactions and algorithmic behaviors of a fully-functional file system [162].

2.4 Summary of this chapter

Timing-accurate storage emulation has a role in the toolbox of storage evaluation approaches. The technique of timing-accurate storage emulation complements currently-utilized techniques where they are unable to satisfactorily explore the use of nonexistent or unavailable storage components in real computer systems. Although the technique is not without its limitations—timing-accurate stor-

age emulation relies both on the availability of accurate storage device models and on hardware and software support for emulator operation—it has applicability in evaluating both currently-available storage products and futuristic storage devices representing significant technological progress in terms of both physical performance and functional capabilities.

The following chapter discusses the design considerations that support the implementation of a timing-accurate storage emulator.

CHAPTER 3
DESIGN OF A TIMING-ACCURATE STORAGE EMULATOR

A timing-accurate storage emulator transparently fills the role of a storage component in a real computer system, enabling application-level evaluations of proposed system designs as if they included the real component. As illustrated in Figure 3.1, a timing-accurate storage emulator must appear to its host system to be the storage subsystem that it emulates. Accomplishing this requires that the emulator correctly mimics the protocols and behavior at the component's external interface: The emulator accepts storage requests from the host system, exchanges data read or written by the host system, transmits request completions to the host system after the correct time elapses, and retains written data to respond correctly to future read requests.

This chapter discusses the design of timing-accurate storage emulators. It introduces terminology to describe storage emulation (Section 3.1), examines an emulator's interactions with the host system and with the experimenter (Section 3.2, page 27), and describes the major functional components that compose a timing-accurate storage emulator (Section 3.3, page 34) and design issues that affect the operational effectiveness of an emulator (Section 3.4, page 38).

3.1 Nomenclature used to describe timing-accurate storage emulation

This section contains terminology that we use in discussing the design and presenting the implementation of a timing-accurate storage emulator. The definitions provided herein are intended to provide an initial understanding of each term, with the full semantics of each term becoming clearer during a thorough examination of their use in the dissertation text.

host system The hardware and operating system upon which the evaluation applications execute during an experiment.

emulation system The hardware and operating system upon which the emulation software executes, which may or may not be the same as the host system (Section 3.2.2).

emulation software The timing-accurate device model and the software infrastructure used to execute the device model during emulation. For storage emulation this includes the data management components that retain written data and the communications management components that interact with the host system (Section 3.3).

emulated device The storage component or components whose timings and behavior are modeled by the emulation software.

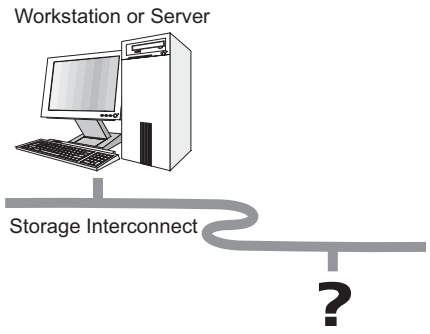


Figure 3.1: Am I connected to a real storage component or an emulator? *The objective of timing-accurate storage emulation is to faithfully replicate the performance behavior of a device model with such fidelity that this becomes a difficult or impossible determination. Compare this illustration with Figure 1.1 (page 3).*

storage interconnect The bus connecting a storage device to the host system. The storage interconnect to an emulated device may or may not be physically available; in the latter case the interconnect timings and behavior must be modeled by the emulation software.

external components Components external in scope to the timing-accurate storage emulator; i.e., any part of the experimental system that is not modeled by the emulation software.

storage request A command from the host system to a storage device to read or write data, to set the device's configurable parameters, or to get the current device status or configuration.

request propagation path The path taken by the storage request arrival notifications (e.g., to the emulated device) and completion notifications (to the host system). In an ordinary computer system, this involves the path through the host operating system, the bus adapter hardware, the storage interconnect, and the hardware and software paths inside the storage device.

3.2 Interactions with a storage emulator

From the perspective of the external system, interactions with an emulated device are identical the interactions with the real device—a storage request is transmitted to the device, an exchange of data occurs with the device, and an acknowledgement is received from the device. Additionally, the system evaluator is involved at a high level with certain internal specifics of the emulator's implementation and operation, including the implications of the hardware platform upon which the emulation software executes, the pre-experimental calibration of the emulator's behavior and initial state, and the post-experimental validation of the emulator's responses to requests. Each of these are discussed in this section.

3.2.1 External system interactions

Conceptually, there are three interactive events between the host system and emulation software that support the execution of each request during an experiment. These are the request arrival at the emulator, the exchange of data, and the request completion by the emulator.

The first interactive event involves transmitting a storage request from the host system to the emulation software. During an experiment, high-level storage traffic—such as application-level file reads and writes—are converted inside the host operating system into a series of individual storage requests to be sent to the emulated device. The structure of these requests depends on the host system's view of the emulated device. If the device is connected to the host using a file- or object-oriented protocol, such as the Network File System (NFS), the structure of the storage requests sent to the emulator will follow the conventions of that protocol; e.g., an object (file) identifier, an offset into that object, and a request length in bytes. If the device is connected using a block-level interface, such as the SCSI interface used in our implementation, the requests will be transmitted in the form of one or more SCSI command blocks; e.g., a device identifier, a block offset, and a request length in blocks. The paths available for transmitting these requests between the device driver in the host's operating system and the emulation software is discussed below in Section 3.2.2. Once the emulation software receives the request, it saves the request's arrival time and issues the request into its internal simulation engine.

The second interactive event involves transmitting the data between the host system and the emulation software in support of the request. The emulation software controls when the data transfer begins. In the case of a READ request, data is transferred from the emulator's internal data cache to the host's OS cache. In the case of a WRITE request, data is transferred from the host's write buffer into the emulation software's data cache. The timing of when this transfer begins is a design option; as discussed in Section 5.1.1, the transfer can either begin immediately once the data are available, or can proceed instead according to timings calculated by the timing-accurate device model. In addition to this event-time management of data transfer, the emulation system may perform extra work in support of data management some time before a request arrives (e.g., prefetching data it predicts will be requested soon by the host system) or after a request completes (paging data between the emulator's internal cache and its backing store).

The third interactive event involves transmitting the completion acknowledgement for the request across the interconnect from the emulation software to the host system. Once the emulator's internal simulation engine determines the completion time of the request—including all device mechanical latencies such as (for an emulated disk) the disk arm seek, media rotational delay, and media transfer time, as well as the communication overheads such as the bus transfer times and device controller latency—the emulation software must monitor the emulation system's clock and initiate its reply to the host system when the appropriate time is reached. However, this completion must generally be delayed whenever the second event's data transfer has yet to start or complete by this time.

3.2.2 Execution domain of a storage emulator

In ordinary systems, storage requests arrive at the device driver on the host system, which then takes care of the low-level details of routing the request to the storage device. This architecture is true of most variants of storage emulation as well. Figure 3.2 and Figure 3.3 show the two most natural points at which to integrate a storage emulator into a host system. The emulation software can either execute locally as a user- or kernel-level process on the host system, or remotely as a process running in a physically separate hardware execution domain from the host system. Under the local emulation approach described in Section 3.2.2.1, 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. Under the remote emulation approach described in Section 3.2.2.2, the host system remains 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 the 3-step interface between the emulation software and the rest of the system described above in Section 3.2.1.

Alternatives to the specific approaches described below may also be appropriate. For example, the entire emulation infrastructure could exist as a user-level library on the host system. Or, a hybrid design allowing greater communication utilizes an intelligent network card to forward SCSI requests to an emulator over a standard network. The availability of hardware and software support for emulator operation, as well as the desired levels of fidelity, resolution, and performance of the emulated device, will together dictate which design options are available to an experimenter.

3.2.2.1 Local emulation

In the case of local emulation (Figure 3.2), the device driver on the host system is modified to be aware of the emulation software process executing on the host and explicitly delivers the request to it instead of to a host bus adapter. Communication between the device driver and emulation software can be efficient; the use of shared memory buffers between the two can reduce memory copy overheads, and consideration of scheduling priorities can minimize undesirable delays on the request's critical path through the system. Our initial investigation into timing-accurate storage emulation was focused on local emulation [72]; the experiments in this dissertation are focused on remote emulation.

The local design allows for easy implementation of out-of-band communication paths between the host operating system and the emulated device. For example, the device driver can measure perceived request service times and easily 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 [111] or exposed eager writes [171]). In addition to device-specific delays, a local emulator must account for bus delays, since there is no

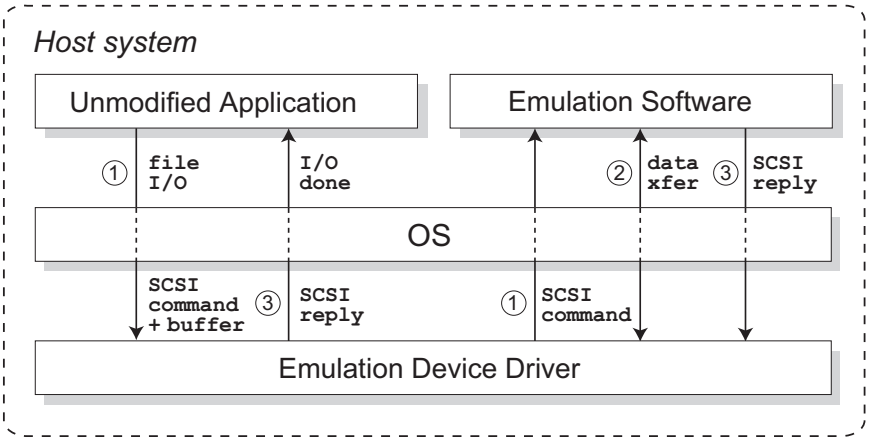


Figure 3.2: Communication paths under local emulation. *As described in Section 3.2.2.1, emulation software communicates directly with a modified device driver in the kernel when run locally. The three external interactions shown here are explained in detail in Section 3.2.1.*

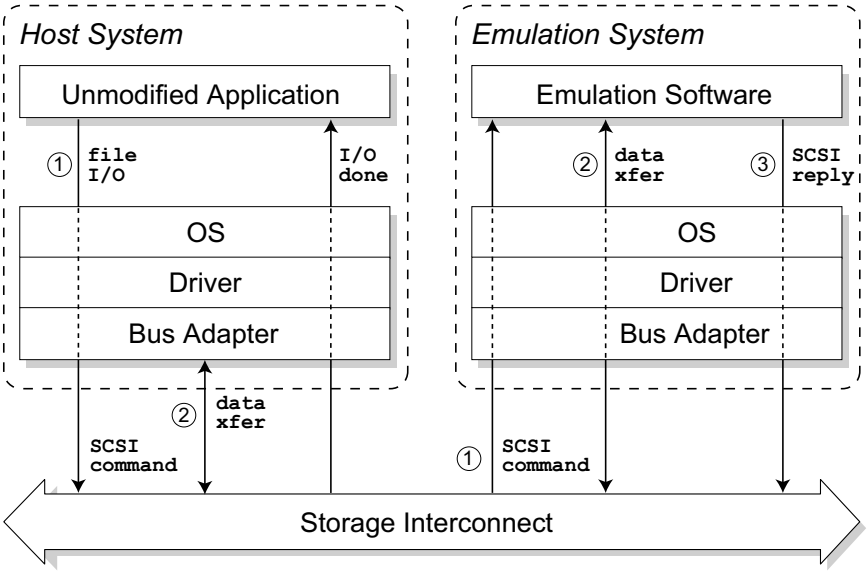


Figure 3.3: Communication paths under remote emulation. *As described in Section 3.2.2.2, all modifications can take place outside the system under test when the emulator is run remotely. This can eliminate any host-system impact caused by the resource requirements of the emulation software.*

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 timings and effects of a new storage interconnect.

The primary disadvantage of local emulation is that, by its nature, it 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 additional memory and one CPU dedicated to emulation can mitigate this overhead, but some interference is inevitable.

3.2.2.2 Remote emulation

In the case of remote emulation (Figure 3.3), the host system does not require a modified device driver; requests are sent unmodified across the storage interconnect to the specially-configured emulation machine, which in turn delivers the request to the emulation software located there. The data transfers and completion message are also sent across the interconnect, just as with a normal storage device, and the unmodified device driver deals with it appropriately.

Depending on the choice of interconnect, it is possible to implement out-of-band communications paths between the host operating system and the emulated device and to evaluate nonstandard device interfaces using remote emulation. For example, in our implementation we achieve this using specially defined SCSI opcodes that would otherwise not be understood by an unmodified SCSI-based storage device. However, these sorts of communication are less robust in remote emulation than in local emulation, and it can be difficult to efficiently feed per-request performance information across the interconnect to the emulation software.

Remote emulation can avoid impacting the host system's performance by performing all emulation functions on separate, dedicated hardware. However, it may not always be the case that system evaluations under remote emulation will use a completely unmodified host system. For example, it may be desirable to place timing and data validation hooks into the host system to verify the proper operation of the emulator, as described below in Section 3.2.4. Or, a special interconnect may be used to connect the remote emulator to the host system as described in the following paragraph.

A remote emulator that is physically attached to the host via a bus need not consider the bus delays in its scope or its calculations, unless it is emulating a different storage interconnect; the bus overheads will actually be incurred during the bus transfers and therefore need not be calculated. It could also be possible to connect the remote emulator using a non-storage interconnect such as gigabit ethernet. In this case, an intelligent network adapter or modified network driver would be used to transfer the data to and from the host operating system. As with the local design, the correct bus transaction overheads will have to be taken into account by the remote emulator.

There are two primary disadvantages to remote emulation. The first is its reliance on special "target-mode" bus hardware on the emulation system—in practice we had access to relatively few SCSI bus adapters that currently support this mode of operation—or alternatively special support

on a now-modified host system to enable a nonstandard interconnect to the emulated device. The second is that the approach gives much coarser per-request timing resolutions than local emulation, because of the hardware involved, and therefore cannot achieve the same throughput and request rates—or perhaps most importantly, minimum request time—as is possible using local emulation.

3.2.3 Pre-experimental calibration

Before a timing-accurate storage emulator can be used for experimental evaluations, it may be necessary for the evaluator to perform several calibrations to ensure maximal emulator accuracy is obtained during experimentation. It is important to focus both on calibration issues that impact the internal device model state and its response to input workloads, and on issues that impact the actual operation and execution of the emulation software.

In terms of the internal device model, the experimenter should define initial conditions for the emulated device state that can be restored deterministically at the beginning of each experimental run. This may involve considerations such as the mechanical state of the emulated hardware (the power-conservation state, the geometric position of the disk arms and platter rotational state) and the state of the software (the contents of the least-recently-used array cache, the remappings of defective media locations). The contents of the emulated device's "media" at the start of an experiment depends on the requirements of the workload; for example, the media could be reset to all zeroes or loaded from a pre-configured image of a valid file system, or restored to match the contents at the end of the previous experiment.

In terms of the emulator's operation, several system factors impact the externally-viewed timing characteristics of the emulator's responses to requests. In general, these factors increase the discrepancy between what time the emulator determines is correct for a request, and what time is actually measured for that request. The factors that impact correctness include hardware factors (delays in bus adapters, interconnect transmission time) and software factors (scheduling delays, data structure management) that may or may not be dependent on individual request characteristics (size, type, interarrival rate). It is important to quantify the effects each of these has on request timings so the emulation software and its internal device model can adjust their calculations and mitigate any timing errors that would otherwise be introduced. The specific choices of which factors to include in the overall emulator calibration depend in part on the configuration of which storage subsystem components are being emulated; for example, if the emulation model does not include a model of the storage interconnect, then the effects of interconnect transmission time should not be considered in this calibration since they are an integral part of each request's propagation path.

For experiments that push the boundaries of an emulator's performance capabilities, it may be important to provide the emulation software with a pre-experimental description of the expected access patterns or request characteristics for the experimental workload. This may be necessary for the emulation software to successfully accomplish its internal memory management of loading and unloading its data cache with the goal of having all read data available when it is needed.

There may be several options for providing this information. For nondeterministic workloads, high-level parameters such as read/write mix, spatial and temporal locality characteristics of requests, or expected request sizes may work well. For deterministic workloads, where the same request stream is generated for each experimental run, it may be possible to provide the emulation software with a description of exactly what requests to expect.

3.2.4 Operational validation

Detailed information about the progress of an experiment using a timing-accurate storage emulator can be gathered during an experiment. This information can then be analyzed and used post-experimentally to provide the evaluator with a quantified metric regarding how accurate the experimental results may be considered to be.

A consideration in the data domain concerns whether the application receives the correct data throughout an experiment. This is most easily addressed using good software implementation practices and a robust implementation testing suite that exercises corner cases and boundary conditions of the emulation software. When a more formal validation is desired, direct or indirect observation can be used to evaluate the experimental-time correctness of the data received by the application. Under the direct observation approach, the actual data transmitted during an experiment (or digests or checksums over the data) are stored and analyzed to verify the correct orderings and contents of the stored and retrieved data. Intermittent spot checks of individual data transfers can be less obtrusively performed, but result in an attendant loss in the completeness of the verification. Under the indirect observation approach, application behaviors during an experiment (such as the standard output of the application; the number, addresses, lengths, ordering or timing of storage requests; or the contents of written data objects) are compared with the behaviors observed during a non-experimental reference run to identify behavioral differences that could have been caused by improper data management.

A consideration in the timing domain involves whether the behavior and performance of the device model was faithfully reproduced in real-time by the emulation software. Successful evaluation of this requires that one or more measurement points along the request propagation path be configured to observe and report on per-request service times at each point. These points should have the characteristic that they are unobtrusive and do not slow down the request (i.e., unduly affect timings on the request's critical path), yet they are powerful enough to collect fine-grained and precise measurements of elapsed time at the point. The measurements from the most appropriate point—that is, the point whose location most closely matches the outermost boundary of the storage subsystem components that are undergoing emulation—can be compared with the times that are being computed by the device model and saved by the emulation software. For pure validation purposes this data can be aggregated and analyzed after the experiment completes; however, these comparisons could also be made during the course of an experiment and fed back into the emulation software in order to achieve more reactive fine-grained tuning of the emulator's behavior.

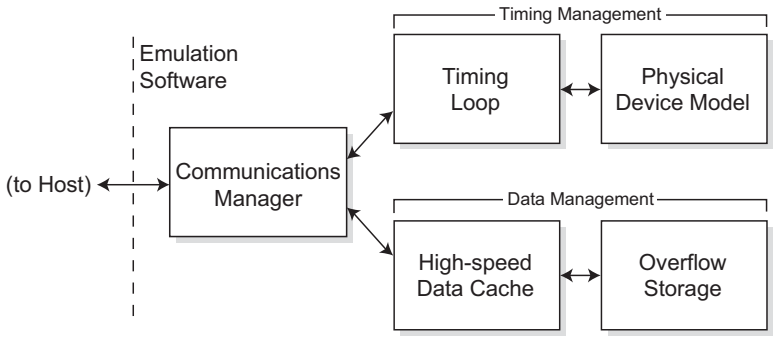


Figure 3.4: Timing-accurate storage emulation software internals. *The five components of a storage emulator provide for the three primary emulator tasks of communications management, timing management, and data management.*

A related but more difficult consideration concerns whether the device model in fact faithfully represents the performance and behavior of a real device. Some model validations are incomplete or exist only over a certain range of inputs; for example, a model that is validated against a disk whose cache is disabled may produce quite incorrect results for when the disk cache is enabled [136]. Ignoring this mismatch raises the possibility that an evaluator will be "led down the garden path" by incorrect evaluation results that otherwise pass the validation tests described in the previous paragraphs. Additionally, some models cannot be validated against real devices, such as our models of MEMS-based storage. Although this issue does not relate directly to the principles or applicability of timing-accurate storage emulation—where the goal is faithful reproduction of the behavior and performance of a device *model* and not necessarily of a device—an emulator can assist in determining the validity of experimental runs. This assumes the author of the device model includes a usable description of under which workload conditions the device model holds valid. During experimental execution the emulator can generate a summary of the workload characteristics, which can later be analyzed against the author's description to produce a quantified metric of "realism" for the experiment.

3.3 Component design

Three essential tasks define the operation of a timing-accurate storage emulator. First, the emulator must correctly support the communication 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 future read requests. Figure 3.4 shows an architecture of five internal components of a timing-accurate storage emulator that together support these three tasks. This section describes how these components work together to satisfy the three tasks of communications management, timing management (the timing loop and the physical device model), and data management (the high-speed data cache and the overflow storage).

3.3.1 Communications management

One function of the emulation software involves the communication of requests and request data: receiving storage requests and WRITE data from the host system, and transmitting request completions and READ data to the host system. The communications management component encapsulates this functionality. To communicate with the host system, this component must understand and export the interface protocols expected by the host system. The communications manager parses incoming requests and passes them to the other emulator components, and it properly formats outgoing messages for return to the host when directed by the timing manager. In addition to servicing requests, the communications manager must respond appropriately to exceptional cases such as malformed requests or device errors.

The communications manager has relatively simple functionality in a local emulator, as it is primarily concerned with two interactions: data transfers between the host operating system and the emulation software—for example, memory management of data buffers shared with the device driver—and system calls that interact with the modified device driver to propagate request arrivals and completions for specific requests. In a remote emulator, the communications manager must understand the specific details of the interconnect used to bridge the host and emulation systems. If the interconnect is a storage bus, such as the SCSI bus used in our implementation, this component must understand how to interact with a target-mode bus adapter[3.1] and the operating system infrastructure that supports receiving and sending storage traffic in target mode. When using a network-based interconnect, it must understand how to interact with the network stack on the emulator system and understand the protocol with the modified device driver on the host system.

Once a storage request is received, regardless of the propagation path used to arrive at the communications manager, this component 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 device such as its capacity, status, or most recent error condition. To fully emulate a device over a given interface, all such commands should be implemented, but in practice a subset of often-used control commands usually suffices. The timing management components notify the communications manager when a request completion is due, after which this component formats the response appropriately for the emulated protocol and forwards the response to the host over the interconnect. Additionally, if the emulation environment is to support an extended storage interface between the host system and emulated device, the specifics of this extra out-of-band or

[3.1] Storage bus adapters ordinarily operate in *initiator mode*, which means they originate (initiate) requests that are directed to storage devices that reside on the bus. Some bus adapter products additionally support *target-mode* operation, where the adapter mimics a storage device (a target) and accepts requests from an initiator. Target-mode operation generally requires special operating system support by the emulation system. Our implementation uses the target-mode functionality provided by two bus adapter products and the FreeBSD operating system in its communications with the host system.

in-band communication path are handled by the communications manager and routed to the appropriate internal component (e.g., the physical device model).

As a concrete example, our implementation is specifically built to interact with the host over a SCSI bus using target-mode bus adapters. When a SCSI command arrives from the device driver, the communications manager parses the command to determine the opcode, starting sector number, and length. For our original implementation based on local emulation, the storage interface also received the SCSI target and logical unit number (LUN) information from the kernel in case more than one device was under emulation. In our remote emulation implementation, this information is conveyed through the SCSI bus protocol. All of these fields are checked for validity and then demultiplexed to the timing and data managers. When invalid opcodes, out-of-range requests, or invalid target/LUN pairs are received, the communications manager generates the appropriate sense code and immediately returns an error condition to the device driver.

3.3.2 Timing management

The two components that together address the timing management task—the timing loop and the physical device model—work together to provide the timing-accurate nature of the emulation. Specifically, the physical device model determines how long each request should take to complete, and the timing loop ensures that completion is reported after the determined amount of time. The concepts that are advanced in this subsection are elaborated upon in Chapter 5.

The timing loop is a real-time loop that monitors the scheduled times of future internal emulator events and compares these times with the advancement of the system clock, inducing other emulator components to take action when the scheduled times arrive. These events represent either internally-visible occurrences, such as the scheduled completion inside the physical device model of the disk arm seeking, or externally-visible occurrences, such as the scheduled notification of the communications manager of a request's completion. This component executes what is conceptually a tight loop, attempting to maximize the accuracy and minimize the variance of the time which other components are notified, but at the same time not hogging the processor or unduly delaying the scheduling and operation of other system components.

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 communications manager to report completion. 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 using an event-based simulator.

An event-based simulator [94] 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. Our implementation uses the DiskSim storage subsystem simulator [24] as the basis for the physical device model.

To synchronize an event-based simulation with real-world time, 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 communications manager.

In practice, the request arrival and completion times may need to be skewed slightly to account for processing and communication delays. As discussed in Section 5.3.2, the arrival time of a request is adjusted backwards slightly to account for the delay in receiving the request. Likewise, the simulator's internal clock may need to run slightly ahead of the real-world clock so that the communications manager will start sending completion messages early enough to account for request propagation delays, such that request completions arrive exactly on time at the host system. 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.

3.3.3 Data management

In most computer systems architectures, secondary storage devices are expected to provide high degrees of integrity and availability for the data that are entrusted to the devices. In normal operation, incorrect data should never be returned in response to a request, and data should be reported as unavailable only in the event of a catastrophic device failure. This holds true for most applications of timing-accurate storage emulation: When an emulated storage device responds to read requests, it is generally expected to transmit data that is consistent with the data that would be provided by a real device—that is, it should return data identical to what was most recently written by the host system to the specified location. Emulation software must therefore retain the data transferred during write requests in a way that enables the correct data to be returned on demand.

The data manager in the emulation software, which is provided for by the high-speed data cache component and overflow storage component, is tasked with tracking all data written by the host system and making that data available to the communications manager in advance of the comple-

tion time for a request. The division of the emulator's real memory resources between the two data management components is a function of how the speed of access to each memory resource compares with the speed of access to the emulated device as determined by the physical device model. We consider resources with equal or faster service times than the emulated device as *high-speed resources*, and those with slower service times as *low-speed resources*. Generally speaking, if the data for a request are not resident in a high-speed resource when a request arrives then the request's completion will be delayed until the data are available. RAM is the primary component we use as a high-speed resource, although our evaluation of disk-based intrusion detection in Chapter 7 uses an actual disk as a high-speed resource. Although none of the experiments reported in this dissertation require the use of low-speed resources, we have considered as future work on data management the exploration of both locally-attached disks and storage resources on network-connected external systems (disks or RAM [9]) as the most readily available and practical candidates for this functionality. Data management is greatly simplified when the emulation system has sufficient high-speed resources to satisfy all requests for a workload, as the emulation software need not be concerned with determining when to promote and demote data between the data cache and the overflow storage.

The timing characteristics of the data manager's operation are much less constrained than those of the timing manager. At the level of individual read requests, successful operation of the data manager is a binary function: either the data are available in a high-speed resource before the time arrives for transmission to the host system, or the data are not available—in which case, the request completion will be unduly delayed until the data are available. When requests are delayed in this manner, the evaluator will need to quantify post-experimentally the overall impact these delays had on the experimental performance and emulated device behavior, and re-run the experiment if necessary.

3.4 Design concerns for correct emulator operation

Several of the design-time options for timing-accurate storage emulators affect the overall validity and effectiveness of emulator-based experimentation. In particular, many of the design decisions—such as the choice of hardware for instantiating the host system and emulation system, or the choice of which storage interconnect is modeled by the emulator—can affect the limits concerning under which operational scenarios emulator-based experimentation remains valid. Evaluators wishing to push the boundaries of an emulator's performance—achieving ever-lower request service times, higher aggregate data throughput, or greater stress on the emulator—must be cognizant of the particular scenarios under which a particular timing-accurate storage emulator design will not work. This section characterizes several of the specific issues that can cause an emulator to fail in its portrayal of a modeled storage device.

Failure due to communications management issues. Two issues involving the design of the communications manager concern the data throughput available at the storage interconnect and the correct management of the interconnect during multi-host operation. Regarding the throughput

available at the storage interconnect, an emulator will be unable to accurately respond to a workload if the workload calls for a higher peak data rate or sustained data rate than the communications manager or the storage interconnect can support. This is especially a concern in an experimental setup where the emulated interconnect is different from the actual interconnect connecting the host system with the target system, both when emulating an evolutionary change (e.g., using a 160 MB/s parallel SCSI bus to emulate a 320 MB/s parallel SCSI bus) or a revolutionary change (using parallel SCSI to emulate serial-attached SCSI or Fibre Channel). Limitations in the supported data rate can be caused by execution issues with the communications software, such as when other emulation software components interfere with the OS-level scheduling of the communications manager, or can be caused by suboptimal use of the existing interconnect—for example, some of our experiments described in Chapter 5 and Chapter 6 use a 160 MB/s parallel SCSI bus, but we are currently only able to configure the Linux-based host system and FreeBSD-based emulation system to support 80 MB/s transfers across this bus during experimentation.

We have not explored emulator operation under a multi-initiator architecture, where multiple host systems are collectively connected to a single emulated storage device. Multi-initiator operation is supported in the specifications of many storage interconnects, and we expect a timing-accurate storage emulator could be used in such a manner. However, support for multi-initiator operation will reduce the design-time flexibility available to the evaluator. Care must be taken to verify that the communications manager can gracefully handle the load of the additional host systems: the communications manager will need to precisely schedule bus transfers to and from each of the host systems to prevent artificial contention among the host systems,[3.2] and the emulation software may need different compensation models for mitigating the per-request errors observed by each of the individual host systems.[3.3] It may not be possible to successfully compensate for per-request errors when the emulator experiences high utilization from multiple host systems, depending on whether the interconnect is included in the scope of the emulated components, because the realization of the request-completion-time error compensation techniques we describe may rely on the flexible scheduling of bus transfers to ensure the availability of the bus for early sending of request completion notifications. In terms of physical dependencies, some hardware configurations are unable to support full multi-initiator operation—specifically, when using the Adaptec HBA for remote emulation in FreeBSD (as described in Chapter 4), only one request at a time can currently be accepted by the communications manager and passed to the emulation software.[3.4] This means that when using the Adaptec HBA in a multi-initiator environment, only one host at a time would be able to send a request to the emulated storage device. An additional (though unlikely) concern is that the increased processing and memory overhead of managing the state of multiple initiators could increase the resource pressure on other emulation software components.

[3.2] This affects the handling of externally-visible intrarequest events, as discussed in Section 5.1.1.

[3.3] Compensation models for per-request errors are discussed in Section 5.3.

[3.4] The reason for this limitation is discussed in footnote 7.2 (page 128); additional software development on target-mode support in FreeBSD is expected to eventually eliminate this problem. Experimental setups using the QLogic HBA under FreeBSD do not share this limitation.

Failure due to timing management issues. Two issues involving the design of the timing manager concern the availability of accurate and efficient device models and the aggregate effectiveness of long-term error compensation. As with any other technique, meaningful evaluations using timing-accurate storage emulation are only possible when representative device models are available for experimentation. Because of the performance-critical nature of storage devices in storage-oriented evaluations, errors in disk modeling can have a substantial impact on overall system performance [115]. Experimentation with faulty device models can lead an evaluator astray, leaving little recourse beyond the exercising of common sense to note unusually good or bad device behavior.[3.5] An additional requirement for models used in timing-accurate storage emulators is efficient model operation, in that the emulation software must complete all processing for a request before the service time elapses for that request. It is this characteristic that currently prevents full-system simulators from executing in real-time. We have not yet been hampered by an inability to satisfy this processing requirement for timing-accurate storage emulators, as the device models with which we are familiar (i.e., those written for the DiskSim simulation environment) complete all per-request processing well faster than real time. However, as models increase in complexity—whether in an effort to achieve more accurate modeling of device behavior, or for emulation-based evaluations of multi-device aggregations such as disk arrays and functional storage bricks—this maximum limit on per-request processing may become a pressing issue.

As discussed in Section 5.3.3, a timing-accurate storage emulator cannot service a request that is shorter than the minimum error along the request propagation path. In our remote emulation experiments, this resulted in minimum per-request service times of approximately 0.1–0.3 ms. Alternatively, local emulation permits much smaller minimum per-request service times; we measured this value as approximately 0.025 ms for a prototype local storage emulator [72]. Additionally, our implementation of the per-request error compensation strategies does not achieve perfect results (i.e., zero error with zero variance), which introduces a hidden assumption that minimizing the per-request error translates to minimizing overall error at the application level. This assumption has the potential of producing misrepresentative experimental results, in terms of long-term drift in error aggregation (e.g., are application and OS-level behaviors biased to include the effects of requests that either take too much time or too little time at the storage device—perhaps by affecting process scheduling or OS-level caching?) or a mismatch between individual and aggregate errors (does a 1% average per-request error actually result in a non-1% overall application-level I/O stall time error?) Although we believe such effects to be minimal, if present, a fuller characterization of the impact of uncompensated-for errors is left as interesting future work.

Failure due to data management issues. An issue involving the design of the data manager concerns the ability of the emulation software to make the correct data available at the necessary rates specified by the workload. This is unlikely to be an issue when the working set of the workload is wholly contained by the high-speed resources; in such cases, it is rather more likely that any defi-

[3.5] Section 8.4.2 discusses the need for better model validation methodologies that would help relieve this burden.

ciencies in the data transfer rate to and from the host system would stem from limitations in the communications management of the storage interconnect than from the data management of the memory cache. However, as discussed in Section 3.3.3, when the working set size is larger than the available high-speed resources it may be necessary to transfer data to and from overflow storage during the emulator-based experimentation. The emulation may fail if the long-term sustained bandwidth to or from the low-speed resources does not match the needs of the workload—especially during attempts to both store data from write requests and retrieve data to serve future read requests—or if excessive computation is required in the prediction code to determine the disposition of current and future data items in the high-speed resources. Section 8.4.1 discusses opportunities for future work in characterizing the extent to which this limits emulator-based experimentation.

Discussion. Many of these limitations are exacerbated by the hardware performance issues inherent to remote emulation, and can be mitigated through the use of local emulation. Alternatively, the degree to which each of these is an experimental limitation can often be relieved through the use of alternative hardware or software components—for example, using the QLogic bus adapter instead of the Adaptec bus adapter in our experimental configuration increases the per-request error introduced along the request propagation path, but allows multiple outstanding requests to be queued at the emulated device, which enabled the experimentation on disk-based intrusion detection systems described in Chapter 7.

3.5 Summary of this chapter

A timing-accurate storage emulator fills the role of an ordinary storage device in a computer system—it accepts requests from a host, transfers the correct request data to or from the host, and transmits request completions to the host at the proper time. To accomplish this, the individual components of the emulation software work together to execute the three fundamental tasks of communicating with the host system, completing storage requests at the times determined by a physical device model, and retaining data written by the host system to satisfy future read requests. When using a timing-accurate emulator in an experimental environment, pre-experimental calibration steps and post-experimental validation actions are available to ensure that the emulator operates properly and responds correctly to the experimental workload.

The following chapter presents the hardware and software framework used in our implementation of a timing-accurate storage emulator.

CHAPTER 4
EXPERIMENTAL APPARATUS

To explore the timing-oriented calibration and validation options for timing-accurate storage emulation, we implemented a functional timing-accurate storage emulator known as *the Memulator*. This implementation, which takes its name from our motivating interest in emulating MEMS-based storage devices, is a remote emulator[4.1] that uses the DiskSim storage subsystem simulator [24] as the basis for its physical device model. The data manager in our implementation allocates a large fraction (approximately 900 MB) of the available RAM memory on the emulation system and uses this RAM as high-speed resources to retain the workload data written by the host system. Our implementation of the timing manager, described in Chapter 5, takes the approach of accurately scheduling the timing of request completions but greedily scheduling the timing of intrarequest events: bus transfers of a request's data are always initiated shortly after the request arrives, which ensures the bus transfer completes early and therefore avoids delaying the transmission of the request's completion notification.[4.2] The communications manager is based on the target-mode user-level software distributed with the FreeBSD operating system, as noted in Section 4.2.

4.1 Experimental hardware

The base experimental apparatus was two workstation-class desktop computers connected by a SCSI interconnect. For most experiments, one of these computers was configured as the host system and the other as the emulation system.

The hardware platform used was a pair of Dell Precision Workstation 340 computers purchased in July 2002, one configured as the host system and the other as the emulation system. Each system contains one 2.0 GHz Intel Pentium 4 processor, with 8 KB onboard L1 cache and 512 KB onboard L2 cache. The host system contains 512 MB RAM attached via two 256 MB RAMBUS inline memory modules; the emulation system contains 1 GB ram via four modules.

Figure 4.1 shows the interconnection of storage devices to the host and emulation systems. The experimental disks used on both systems are 37 GB Seagate Cheetah 36ES SCSI-3 disks, model ST336706LC, revision 0109, whose specifications are shown in Table 4.1. These experimental disks are connected via external 68-pin LVD SCSI cables to Adaptec 29160 Ultra160 SCSI adapters, which in turn attach internally to the PCI bus of each system.

The host and emulation systems are connected by both a 68-pin LVD SCSI cable and a point-to-point Fibre Channel cable for emulation experiments. To support this interconnection, each system

[4.1] The approach of remote emulation is discussed in Section 3.2.2.2 on page 31.
[4.2] Scheduling options for intrarequest events are discussed in Section 5.1.1 on page 46.

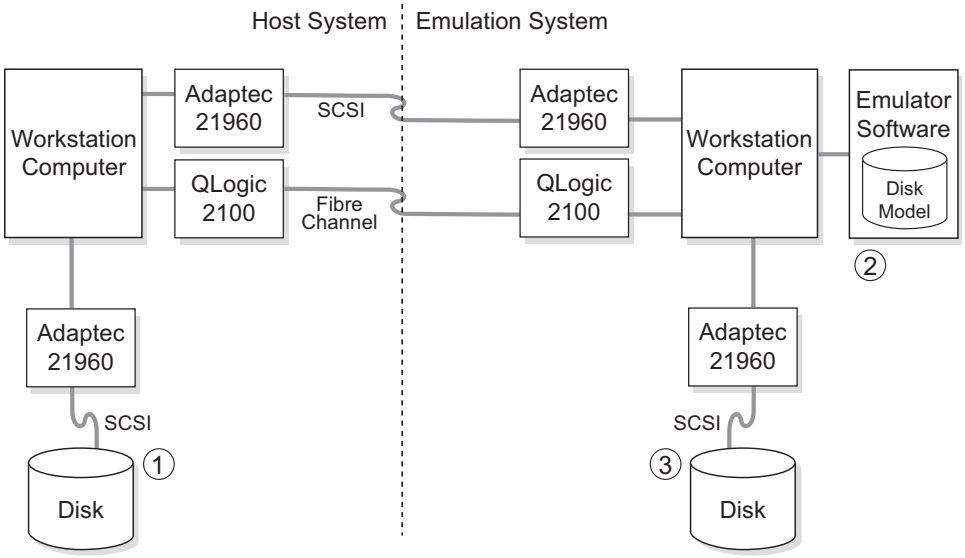


Figure 4.1: Hardware configuration for prototype emulation experiments. *Depending on the requirements of the experiment, the timings seen at the host system for the emulated device may come from three sources: (1) a local disk on the host system, (2) the physical device model in the emulation software, or (3) a local disk on the emulation system.*

contains an additional Adaptec 29160 Ultra160 SCSI adapter (distinct from the identical adapter connected to the experimental disks) and a QLogic QLA2100 Fibre Channel adapter, each of which support target-mode SCSI operation.

The local disks used for system files on each system are Fujitsu Enterprise SCSI-3 disks, model MAN3184MP, revision 5507, whose specifications are also shown in Table 4.1. To remove any dependence of the experimental results on the choice of local disks, these disks are connected via each Dell's onboard Adaptec 29160N Ultra160 SCSI adapter, which is not otherwise used in support of experimentation.

4.2 Configuration of the communications manager

The host system uses the Linux 2.4.20 operating system, based on a modified Red Hat Linux release 8.0 distribution. The emulation system uses the FreeBSD 5.2-RELEASE operating system, with the special configuration options to support target-mode operation shown in Table 4.2.

The communications manager of the emulation software is based on the SCSI target-mode example software included with the FreeBSD 5.2 distribution (and earlier 5.x and 4.x distributions) in the directory `/usr/share/examples/scsi_target`. This software includes both user-level and kernel-level routines and together is used to drive the Adaptec and QLogic host bus adapters in target mode. The current version of the FreeBSD target-mode infrastructure and example software was contributed by Nate Lawson. Certain of the earlier 4.x releases of FreeBSD include an initial version of the `scsi_target` software that was contributed by Justin Gibbs.

	Seagate Cheetah ST336706LC	Fujitsu Enterprise MAN3184MP
Year of introduction	2001	2001
Formatted capacity	33.9 GB	17.1 GB
512 B sectors	71,131,721	35,885,448
Interface	Ultra160 SCSI	Ultra160 SCSI
External transfer rate	160 MB/s	160 MB/s
Spindle speed	10,000 rev/min	10,025 rev/min
Average rotational latency	2.99 ms	2.99 ms
Average seek (read, write)	5.2 ms, 6.0 ms	4.5 ms, 5.0 ms
Track-to-track seek (read, write)	1.0 ms, 1.5 ms	0.4 ms, 0.6 ms
Internal transfer rate	63.2–49.1 MB/s	84.1–52.0 MB/s
Internal buffer	4 MB	8 MB
Disk platters, heads	2, 4	1, 2
Platter size (OD)	95 mm	84 mm
Cylinders	19,036	30,200
Acoustic noise (idle)	3.2 Bels	3.6 Bels
Power requirements (idle)	9 W	7.8 W
Mass	0.81 kg	0.75 kg
Form factor (H×W×D)	26 mm × 102 mm × 147 mm	

Table 4.1: Specifications for the Seagate ST336706LC and Fujitsu MAN3184MP disks used in experimentation. *These values are not reported for comparison but rather for reference. Both products are used as experimental disks in the emulation system. The Seagate product is also used as an experimental disk in the host system, and a physical device model of the disk is used in the emulation software. Further details on each disk are available in the product manuals* [53, 148].

```
device   targ                 # SCSI target emulation device
device   targbh               # Default target device when not driven
device   ispfw                # Downloadable firmware for QLogic QLA2100
options  AHC_TMODE_ENABLE=0x2 # Enable Adaptec 29160 target-mode
options  ISP_TARGET_MODE=1    # Enable QLogic QLA2100 target-mode
options  VFS_AIO              # Enable asynchronous I/O for data manager
options  CAMDEBUG             # Enable verbose debugging of SCSI paths
```

Table 4.2: Configuration parameters for target-mode emulator operation under the FreeBSD 5.2-RELEASE operating system. *We compiled a custom operating system kernel using these parameters to enable target-mode support for the Adaptec and QLogic adapters and asynchronous I/O support for the data manager's read operations.*

4.3 Alternative configurations

In addition to various bus adapter products from Adaptec, Inc., and QLogic Corporation, target-mode bus adapters are available from Advanced Storage Concepts (ASC), Inc., and from LSI Logic Corporation. ASC offers supported target-mode software drivers for the Microsoft Windows platform as its core "VirtualSCSI" product, and the company reports that hardware and software support for target-mode Fibre Channel, Serial Attached SCSI, and iSCSI operation are scheduled to be available from ASC in late 2004.

An alternative target-mode software implementation for FreeBSD was written by Jeff Fellin at Bell Laboratories as part of the StarFish project [54]. This implementation was made available on the Internet at `starfish.bell-labs.com`. The StarFish target-mode code was inspired by Gibbs' `scsi_target` examples in the early 4.x releases of FreeBSD, and therefore may not work seamlessly with the updated kernel-level target-mode code in newer FreeBSD releases.

Limitations in the design of the Linux SCSI subsystem preclude easy integration of target-mode operation into the Linux kernel. A project at the Interoperability Lab at the University of New Hampshire aims to rewrite portions of the Linux SCSI mid-layer to support flexible target-mode operation [121]. A recent update to this codebase by Vladislav Bolkhovitin was announced on the Linux kernel mailing list on June 16, 2004.

Support for target-mode SCSI operation has been documented in other operating systems, including Solaris from Sun Microsystems and AIX from IBM Corporation. A good starting point for designing an implementation from scratch is Brian Sawert's discussion of target-mode SCSI support in Windows [137, pp. 127–160].

CHAPTER 5
OBTAINING ACCURATE TIMINGS FOR EMULATED STORAGE REQUESTS

This chapter develops techniques for maximizing the timing accuracy of a timing-accurate storage emulator, including executing an emulated storage device model in real time (Section 5.1), instrumenting an emulator to measure internally- and externally-observed request timings (Section 5.2, page 50), quantifying errors in the measured timings (Section 5.3, page 56), and calibrating an emulator to mitigate errors by modifying the emulator's behavior based on knowledge of measured request times (Section 5.4, page 66).

5.1 Tasks involved in timing management

In the timing domain, the objective of a timing-accurate storage emulator is to respond to requests externally with the exact timings associated with the device it models. In theory, this means that a host system would be unable to distinguish, using solely an analysis of request times as a comparator, which storage component is being used when it is connected to either the real storage device a timing-accurate emulator of that device (as illustrated previously in Figure 3.1 on page 27). Practically, this means that the timing-accurate storage emulator must complete each storage request in real time as accurately and precisely as possible with respect to the device simulation model running inside the emulator.

5.1.1 Timing-accurate execution of a storage device model

At a high level, an external entity provides the storage emulator with a start time (triggered by the arrival of a new storage request) and the emulation software will respond in accordance with the consideration of a request deadline (after an interval corresponding to the overall service time). This is similar to the requirements of a real-time environment, but with a significant difference: the "goodness" of an emulated storage request is measured continuously in terms of how closely it matches a desired request time; in a real-time system, goodness is a discrete measurement of whether the request completed before a deadline or not. This difference is illustrated in Figure 5.1.

The request response times are determined internally to the emulator by applying the incoming request stream to a model of the desired device. This model can range in complexity and detail from a simple constant-time software generator, to a detailed analytic engine, to a low-level hardware- and firmware-inclusive device simulation program, to a hardware-accurate device prototype. A core requirement for software-based models is that the amount of processing time required for any given request is less than the post-queueing service time for that request; otherwise, the emulator would miss the deadlines for any request that required extra processing time.

To achieve this, it is possible to have multiple degrees of fidelity achieved by the device model, depending on the accuracy desired during a particular emulation phase and the amount of processing required to achieve finer-grained results. The fidelity can be varied during an experiment, or it can be varied once the experimental requirements are known in order to meet the performance goals required by the experiment. This idea of dynamically adjustable simulation detail was used effectively in experiments using the SimOS full-system simulator [79, Ch. 4]. Taking the idea one step further, an emulator could use levels of fidelity on a per-request basis. This would involve first making a simple guess of the request time and then attempting to iteratively refine that guess through additional processing until such time as the originally guessed interval elapses.

The handling of externally-visible intrarequest events (such as device-initiated bus arbitration activity and data transfers with the host system) can optionally be treated in a timing-accurate manner by the emulation software, as illustrated in Figure 5.2. The detail to which this is possible depends on several factors, including the scope of which components are being emulated, the limitations of the software control of the bus adapter hardware, and the level of request detail provided by the device model. An interesting question for future investigation involves identifying under which circumstances it is necessary for such events to be timing-accurate—for example, determining whether the host system bus adapter performance is adversely affected when request completions are delayed for several milliseconds beyond the final data transfer for a request (as in Read case B of the illustrated figure). In our implementation of a timing-accurate storage emulator, bus transfers are always initiated shortly after a request arrives.

As discussed in Section 3.2.4, an important characteristic of the work presented herein is that the correctness of an emulator is independent of the correctness of the simulation model it follows. The question of executing a device model in real time, and observing and validating an emulator's behavior in comparison to the model, is divorced from the question of whether or not a particular device model has itself been validated. However, the real-world representativeness of an emulation experiment depends on the accuracy of the model; it is important for an evaluator to keep this in mind to avoid being misled by misrepresentative results.

5.1.2 Compensating for emulation-induced request timing errors

In the timing domain, the metric of success for a timing-accurate storage emulator is the degree to which it remains faithful to the device model timings when responding to individual requests. In order to measure this degree of effectiveness, it is necessary to first measure individual request times throughout the system and then correlate the measurements to determine and mitigate any future errors. Per-request errors may stem from hardware-induced request propagation delays (e.g., delays in bus adapters or transmission time) or execution issues inside or outside the emulation software (data structure management, process scheduling delays). Measuring and correlating request times enables an emulator to adapt to these errors through pre-experimental calibration, operational monitoring, and post-experimental verification of the desired emulator performance and behavior.

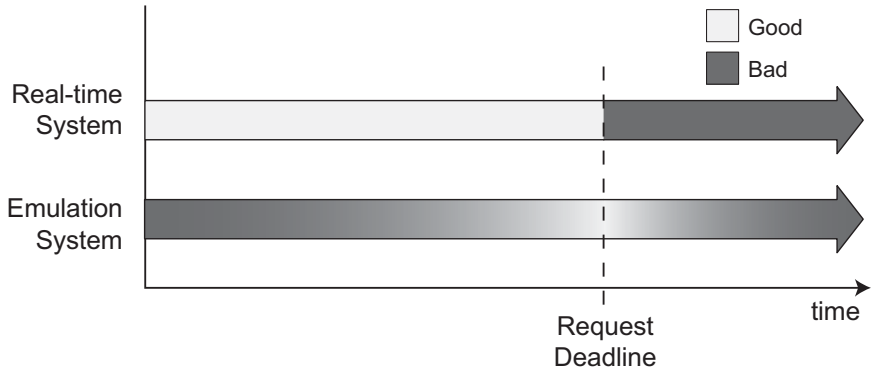


Figure 5.1: Success metrics in a real-time environment and an emulated environment. *An emulated environment is similar to a real-time environment, but with a subtle difference. The metric of goodness in an emulated environment depends on the distance between the request service time and the deadline; in general, it is equally good (or bad) to be one time unit too early as it is one time unit too late. The metric in a real-time environment is binary, in that it is good if the request completes before a deadline and bad otherwise.*

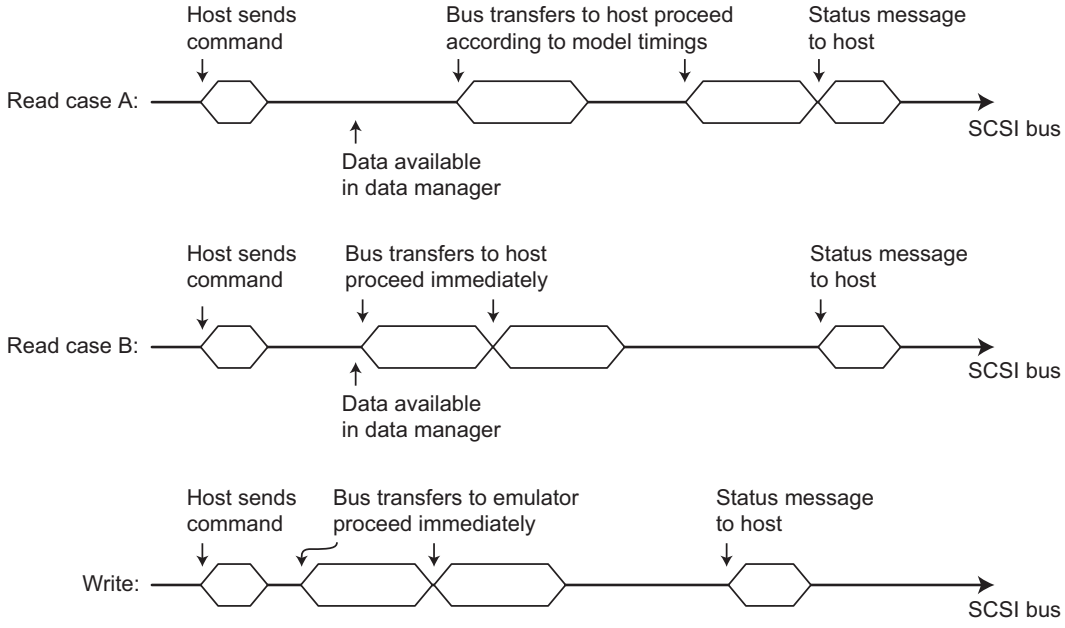


Figure 5.2: Degree of emulated detail for storage requests. *When responding to read requests, a timing-accurate storage emulator can faithfully reproduce the intrarequest timings reported by the physical device model (such as the exact time bus transfers are initiated) as shown in Read case A. Alternatively, the emulator can faithfully reproduce only the timings of request completions as shown in Read case B; this allows greater flexibility in operation of the communications manager and data manger. When responding to write requests, the data transfers are initiated immediately once the physical device model reports that a write buffer is available; this represents our assumption that most storage devices operate thusly in response to write requests.*

There are several places in a computer system where times can be measured for eventual feedback to an emulator. Certainly, the times from the device model itself will need to be recorded and tracked, as most other comparisons will be made against the model's times. Other locations that are candidates for measurement include inside the emulation subsystem itself, inside the operating system device driver (or a similar point inside the OS-level communication path) on the emulation and host systems, and at the application level on the host system (using either a direct bus-level interface or the standard file system interface). It may be desirable to take measurements at some or even all of these locations. In the case of local emulation, where the host and target are on the same physical hardware, it remains useful to distinguish timepoints inside the operating system that are inside the emulation infrastructure and outside the emulation infrastructure. Alternatively, in the remote emulation case where the bus itself isn't under emulation, it may not be necessary to take time measurements on the host system at all, eschewing these in favor of additional measurement points inside the target system's OS.

Analysis of the collected data can occur before, during, or after an experiment, depending on the manner (static or dynamic compensation, described below) in which the data are fed back into the emulation environment. Data analysis before or after an experiment has the advantage of minimizing the effect of the analytical overheads on the run-time behavior of the system, with the limitation that the emulator cannot be tuned during the course of an experiment based on the observed request errors. To combine the advantages of the approaches, it may be useful to perform this analysis at two or perhaps all three times: simpler analyses can be done during the course of an experiment, with more detailed studies done before and/or after for better evaluation of the experimental results and better tweaking for future experiments. An additional dependency concerning the decision on which approach to take is that there needs to exist a method for aggregating times to a central place for this processing; it may be difficult to feed times measured on the host system back into the emulator in a remote emulation environment, limiting the options of what analyses can be performed.

Timing measurements can be used internally by the emulation software using an approach based on static compensation or dynamic compensation. Static compensation, where emulation parameters (such as $\Delta T_{lookahead}$ and ΔT_{skew}, both introduced below in Section 5.3.2) are adjusted a priori to an experiment, is useful for the initial calibration of an emulator. An important aspect of the static compensation approach is to ensure that the training workload used for the static analysis is representative of the workload that will be used during the actual experiments. Dynamic compensation is where the emulation parameters are adjusted during an experiment in an effort to reduce errors evident in the timing measurements. This may be especially appropriate for host system workloads that operate in multiple distinct phases, with each phase providing a unique load on the storage subsystem, when the emulator performance and behavior might be affected by such factors as the request interarrival rate or the sizes of individual requests in each phase. The compensation may be able to make use of information both concerning the accuracy of requests, i.e., whether the average error between the measured request times and the device model times is low, and concerning the

precision of requests, i.e., whether the variance of this error is minimal. It is also important to evaluate the emulation infrastructure in terms of the maximum possible performance: for example, what is the maximum sustained request rate for the emulator during bursty request traffic without missing deadlines, or how long can the emulator sustain a particular data transfer rate without missing deadlines.

Additionally, there may be resource allocation concerns that relate to the environment of timing-accurate storage emulation. For example, in the case of local emulation it is desirable to measure the effect the emulation infrastructure has on the host system (e.g., the amount of processing time consumed by the emulator; the memory footprint and backing-store bandwidth required for effective emulator operation), to quantify any deleterious effect the emulator may have on the experimental workload running on the same system. In general, the emulation system should be designed to make efficient use of its processing resources in an effort to maximize both the accuracy and precision for any given workload, as well as to maximize the range of potential workloads that can be supported by the emulator.

5.2 Collection and comparison of observed request response times

The basis of our timing-specific emulator calibration and experimental validation involves measuring the total response time for each storage request. These response times are measured at multiple observation points along the request propagation path in order to gain an understanding of sources of delay and error introduced by the emulation infrastructure. Because response time includes both queueing time and service time, and because our implementation does not require a finer-grained measurement than response time itself, we practically measure response time by taking timepoint measurements both when a request first arrives at the measurement point and when the completion notification propagates up to the measurement point.[5.1] The locations where response times are collected in our implementation are described in Section 5.2.1. Analysis of these collected request response times to quantify the location and magnitude of errors is discussed in Section 5.2.2.

Our experiments implement two types of metrics for evaluating the degree to which externally-observed request times match the device model's internal times: individual request comparisons (Section 5.2.3) and aggregate request comparisons (Section 5.2.4). These metrics are useful for identifying inefficiencies and other sources of experimental error in the system when designing and implementing a timing-accurate storage emulator. As discussed in Section 5.1.2, these metrics can be used to tune emulator behavior to best accommodate a particular request stream during or after an experiment. Additionally, these metrics are useful for post-experiment evaluation of the representativeness of experimental results when using a timing-accurate storage emulator.

[5.1] Because these timepoint measurements ("time stamps") are taken multiple times in several locations during each storage request, and taking into account the possible sub-millisecond resolution of an emulated storage request, the facility for taking time stamps must be both efficient and have a small resolution. Our technique for taking efficient and precise time stamps is presented in Appendix C. These timepoint measurements are translated into an interval of elapsed time (ΔT) using Equation C.2 on page 169.

5.2.1 Measurement points for collecting response times

In our prototype implementation, we measure request response times at three measurement points ($M = 3$) along the request propagation path: inside the host system OS (MP_1), at the interface between the emulation software and the emulation operating system (MP_2), and inside the device model itself (MP_3).

Measurement point MP_1. To measure timepoints inside the host system OS, we built a Linux loadable kernel module that interposes custom software along the request propagation path in the Linux SCSI subsystem. Specifically, our module intercepts the `queuecommand()` call with a custom function that serves as a new bridge point from the SCSI mid-layer to the device-specific driver, where the first timepoint is taken. After the timepoint is taken, the original `queuecommand()` function is invoked. Additionally, our module replaces the mid-layer `scsi_done()` function with a custom function that takes the second timepoint before invoking the original `scsi_done()` function. Each timepoint is buffered temporarily in kernel memory, in order to prevent further interference with the request's critical path, and is eventually written out to a file along with the two-byte SCSI opcode, the block offset and block length of the request, and a flag denoting whether the timepoint represents the arrival or completion of the request.

Measurement point MP_2. The interface between the emulation software and the emulation system OS consists of reads and writes of CAM Control Blocks (CCB) on an open file descriptor representing a special target-mode control device in the OS. The emulation software is notified of request arrivals (and other events, such as bus transfer completion notifications) from the OS when a specially marked CCB is passed up from the OS during a `read()` system call by the emulation software, at which point a timepoint is taken by the software. When the emulation software determines that a request completion notification is sent, another marked CCB is sent to the OS during a `write()` system call; the software takes a timepoint immediately preceding this system call. Both timepoints are temporarily buffered and are eventually written to a file along with the SCSI opcode, the block offset and the block length of the request, a flag denoting whether the timepoint represents the arrival or completion of the request, and an internally-generated request serial number.

Measurement point MP_3. Authoritative request service times are computed by the physical device model during ordinary emulator operation. These times are based on the value of the physical model's internal clock at the time of request arrival and request completion. This internal clock is kept loosely synchronized to the emulation system's clock through its interactions with the timing loop. The physical device model used in the Memulator operates on a clock of the elapsed time since the start of the simulation (referred to historically as "simtime," the current simulation time). Simtime is measured by applying Equation C.2 (page 169) with t' equaling the current processor cycle count and t'_i equaling the cycle count at the start the simulation. ΔT is evaluated simply as the difference between the arrival simtime[5.2] and the completion simtime. This value is temporarily

[5.2] In practice, we generate the arrival timestamps for MP_3 response time calculations by reading the timing loop's wall-clock time when the simulator is invoked, and intentionally skewing this value to account for delays in the request

buffered and is eventually written to a file along with the block offset and block length of the request, a flag denoting whether the request was a read or write, and the request serial number generated earlier by the emulation software.

Each measurement point creates a separate file containing the measured timepoints and associated request data for each timepoint. These files are processed and collated into a single file containing correlated timepoint data for each observed request. Correlations between the emulation software timepoints and device model elapsed times are simplified via the shared serial number generated by the software.[5.3] These requests are then correlated with the host-observed requests by matching the completion order of requests in the two timelines.

5.2.2 Quantifying sources of error in emulated requests

In order to adjust the behavior of the emulation software and effect the exact desired times at the external observation point, it is first necessary to form an understanding of what factors contribute to the incorrectness of the propagated times. It should be noted that this does not (necessarily) involve quantifying known or unknown sources of error inherent in the device model itself; rather, this question deals with the inherent propagation delay as requests travel through the emulation system.

The device models used in the Memulator include details of a device's mechanical operation and the associated device firmware overheads. These models may additionally include a model of the host-to-device interconnection bus, which was the case for our experiments. We therefore chose the measurement point located inside the host operating system (at the Linux SCSI mid-to-low-layer interface, as described above in Section 5.2.1) as the target point for correctly matching the device model times, as this point is closely located to the actual bus-level interface on the host system. Our analysis is broken down into the two propagation paths between the three measurement points: first, the path from the host system OS to the system-call interface between the emulation software and the emulation system OS; second, the path from the system-call interface to the device model being executed by the timing manager.

The path between the host system OS and the emulation software consists of the interconnection bus itself, the host and target bus adapters, and the low-level device drivers on the host and target systems. We have identified two components that contribute to request propagation delays across this path. The first is a roughly constant-time delay, independent of a request's size or type, between the time the request completion notification is initiated by the emulation software and the time it is received by the host system. This constant-time delay can be a noticeable fraction of the overall

propagation path (as per the discussion in Section 5.3.2.1). The completion timestamps are solely based on the simulator's unmodified simtime at request completion time.

[5.3] This requires more careful logic when shared serial numbers are not available. Because a request (R_1) can be delayed by the emulation software (for example, because the communications manager not having completed the transfer of data for a request, or because the data manager not having data available in high-speed resources when needed), the communications manager might complete a subsequent request (R_2) before it can complete R_1. In this case, the ordering of request completions seen at MP_2 will not match the ordering specified by MP_3. We noticed this occur occasionally in our implementation.

request response time; we measured average times of 0.099 ms using the Adaptec HBA over a SCSI interconnect and 0.282 ms for the QLogic HBA over a Fibre Channel interconnect. Although we have not identified the individual contributing factors that comprise these fixed delays, we suspect they are rooted primarily in the arbitration time required for each HBA to gain control of the bus, interrupt dispatch delays, and execution of the device firmware[5.4] and target-mode device driver software. The second delay-inducing component is the bulk transfer time required to transmit the requested data across the interconnect at wire speeds. Because our design focuses only on achieving proper request response times and not on the detailed accuracy of bus transfers initiating at a correct, modeled time, our implementation initiates data transfers immediately once the data are available. As these data transfers generally complete before the request notification is to be sent, we are able to omit the bus transfer time as a consideration in our calculations. However, our system must take care to note whenever a request notification was unable to be sent on time due to the bus transfer not yet having completed.

The errors introduced by the emulation software are inherently related to the efficiency of the software implementation. In previous versions of our emulation software, we detected overheads that were dependent on both the type and size of the request. In our current implementation, which represents a from-scratch rearchitecting of the individual emulator components, the per-request overheads have been reduced to a constant error of 0.05 ms per request with no dependency on the request type or request size. Much of the earlier overheads in our implementation were due to inefficient communication between the timing, data, and communications managers. Substantial delays were caused by data copies between heavyweight processes that relied on the interprocess communication facilities provided by the OS. By utilizing shared memory buffers between the components, and simplifying the timing and communication managers into a single process to more quickly propagate request completions to the target OS, the overall efficiency of our implementation was greatly improved.

The overall expected per-request error between the primary selected measurement point and the device model can then be specified in terms of the sum of the individual component errors between the M measurement points. This is demonstrated in Equation 5.1, with $E_{i\rightarrow j}$ representing the error between measurement points i and j:

$$E_{1\rightarrow M} = E_{1\rightarrow 2} + E_{2\rightarrow 3} + \cdots + E_{(M-1)\rightarrow M} \tag{5.1}$$

Our three measurement points result in $M = 3$ for our implementation. The empirical analysis we used to determine $E_{1\rightarrow 2}$ (the error introduced between the host and the emulation software) and $E_{2\rightarrow 3}$ (the error introduced between the emulation software and the device model) is presented below in Section 5.4.

[5.4] The Adaptec and QLogic bus adapters are certainly good products, but it would surprise the authors if much development effort has been spent optimizing target-mode operation (compared with the certain optimizations that have been performed for the normal initiator-mode operation). Although target-mode storage operation is not an unknown practice, it has not historically been a market-driving force.

5.2.3 Individual request comparison metrics

An individual comparison metric quantifies the accuracy of an emulator's responses on a per-request basis. Time stamps for each request are collected at various points in the system during the propagation of events along the request path (i.e., the request arrival and completion notifications). An individual metric evaluates the differences in the time stamps for each request to identify sources of inaccuracy in the request path and to quantify the correction factors that can be used to mitigate the inaccuracies. These differences are then summarized across all requests and reported using both quantitative values and a histogram.

Our metric compares the ΔT (see Equation C.2) for the two quantities we refer to as the reference time and the comparator time. The *reference time*, ΔT_R, is the supposed "correct," actual response time of the request—for example, when one of the two measurements concerns the device model's internal times, it would be appropriate to treat this device model time as the reference time. Note, however, that non-device-model times may also be used as the reference time. $\Delta T_R(n)$ represents the reference time for storage request n, where $1 \leq n \leq N$, with N representing the number of requests. The *comparator time*, ΔT_C, is the time being compared to the reference time. In general, the comparator time should be at the closer of the two measurement points to the host system. (Another way to visualize this is that the reference time should be farther from the host system along the request path & closer to the device simulation model.) When evaluating the metric over the measured time in the host system OS and the communication manager in the emulation software, the appropriate assignment for the emulation software would be the reference time.

For all experiments, we report N as well as the mean reference time, $\overline{\Delta T_R}$, which is calculated by Equation 5.2. The product of $\overline{\Delta T_R} \times N$ provides a measure of the overall storage subsystem time for an experiment. $\overline{\Delta T_R}$ is important for interpreting the relative magnitude of the absolute quantities below.

$$\overline{\Delta T_R} = \frac{\sum_{n=1}^{N} \Delta T_R(n)}{N} \tag{5.2}$$

We suggest a minimum of two quantities be reported as individual comparison metrics. The first quantity is the mean error between the reference time and the comparator time. This quantity, which we refer to as EI_{avg} ("error individual average"), is calculated by Equation 5.3. The magnitude of EI_{avg} indicates the degree to which positive errors (the comparator time was larger than the reference time) cancel out negative errors over the life of an experiment; for experiments with small values of EI_{avg} the total time consumed by the emulated storage subsystem was correct, but the individual requests may or may not have been accurate. We additionally report $EI_{avg\%}$, calculated by Equation 5.4. $EI_{avg\%}$ relates the magnitude of the absolute error in terms of its relative effect on the experimental result.

$$EI_{avg} = \frac{\sum_{n=1}^{N} \left[\Delta T_C(n) - \Delta T_R(n) \right]}{N} \tag{5.3}$$

$$EI_{avg\%} = \frac{EI_{avg}}{\overline{\Delta T_R}} \times 100.0 \tag{5.4}$$

The second quantity is the root-mean-square error between the reference time and the comparator time. This quantity, which we refer to as EI_{rms}, is calculated by Equation 5.5. EI_{rms} complements the measure of EI_{avg} by indicating to what degree the individual requests were accurate: When EI_{avg} is small, small values of EI_{rms} indicate that the requests were mostly (or completely) accurate. When EI_{avg} and EI_{rms} are similar, that means that there was generally a constant error between the reference and comparator times. When EI_{rms} is large, there were several (to many) requests that had a large, variable error between the reference and comparator times. As before, we additionally report the root-mean-square percentage error. This quantity, which we refer to as $EI_{rms\%}$, is calculated by Equation 5.6. As with the average per-request percentage error above, $EI_{rms\%}$ relates the absolute magnitude of EI_{rms} in terms of its relative per-request impact on the experiment.

$$EI_{rms} = \sqrt{\frac{\sum_{n=1}^{N} \left[\Delta T_C(n) - \Delta T_R(n) \right]^2}{N}} \tag{5.5}$$

$$EI_{rms\%} = \frac{EI_{rms}}{\overline{\Delta T_R}} \times 100.0 \tag{5.6}$$

In addition to reporting EI_{avg} and EI_{rms}, we visually display the distribution of timing inaccuracies by plotting histograms of the individual request errors, $\Delta T_C(n) - \Delta T_R(n)$, and of the individual request percentage errors, $[\Delta T_C(n) - \Delta T_R(n)] \div \Delta T_R(n) \times 100.0$, for each request ($1 \leq n \leq N$). The request error histograms in this dissertation use a bucket size of 0.02 ms, and the percentage error histograms use a bucket size of 1%.

5.2.4 Aggregate request comparison metrics

An aggregate comparison metric quantifies the accuracy of an emulator's responses for all requests over the lifetime of an experiment. An aggregate metric does not compare measured ΔT values for an individual request, but rather compares at the distribution of ΔT values at an individual measurement point with the distribution at an alternate point. The "demerit figure" aggregate metric advanced by Ruemmler and Wilkes for evaluating storage device models against real devices [136] is the basis for our approach to aggregate evaluations herein.[5.5]

[5.5] Specifically, Ruemmler and Wilkes plot the time distribution curves from two measurement points and use the root-mean-square of the horizontal distance of the curves as their metric; our approach is identical to this. EA_{rms} and $EA_{rms\%}$

Our metric performs an ordered comparison of the ΔT values across two distributions, the reference distribution and the comparator distribution. The *reference distribution*, D_R, contains the reference times—ΔT_R, described above in Section 5.2.3—sorted in increasing time order. $D_R(1)$ is the smallest reference time; $D_R(n)$ represents the nth smallest reference time, where $1 \leq n \leq N$, with N representing the number of requests and therefore the number of reference times. The *comparator distribution*, D_C, is a sorted list of the N comparator times (ΔT_C). Comparisons are made between elements of the same position within the two distributions; i.e., $D_R(1)$ is compared with $D_C(1)$. No importance is attached to which request produced a particular value, as the pair of times represented by $D_R(n)$ and $D_C(n)$ may or may not have been measured for the same request.

For all experiments, we report N as well as the mean reference time, $\overline{\Delta T_R}$, which is calculated by Equation 5.2. The product of $\overline{\Delta T_R} \times N$ provides a measure of the overall storage subsystem time for an experiment. $\overline{\Delta T_R}$ is important for interpreting the relative magnitude of the absolute quantity below.

Following Ruemmler and Wilkes' example, we report the root-mean-square error between the reference distribution and the comparator distribution as the aggregate comparison metric. This quantity, which we refer to as EA_{rms} ("error aggregate root-mean-square"), is calculated using Equation 5.7. An EA_{rms} of zero indicates that the every value found in the ΔT_R has a one-to-one correspondence with an identical value in ΔT_C, but in such a case it does not necessarily follow that the entries in ΔT_R are ordered the same as in ΔT_C. We additionally report the root-mean-square percentage error. This quantity, which we refer to as $EA_{rms\%}$, is calculated by Equation 5.8. $EA_{rms\%}$ relates the absolute magnitude of EA_{rms} in terms of the mean request time.

$$EA_{rms} = \sqrt{\frac{\sum_{n=1}^{N} \left[D_C(n) - D_R(n) \right]^2}{N}} \tag{5.7}$$

$$EA_{rms\%} = \frac{EA_{rms}}{\overline{\Delta T_R}} \times 100.0 \tag{5.8}$$

In addition to reporting EA_{rms}, we visually display the reference and comparator distributions by plotting the cumulative distribution function for each distribution on a combined graph.

5.3 Request response generation inside an emulator

Once a facility is in place for collecting and measuring request response times, the emulation software may be tuned in an attempt to mitigate the per-request errors caused by the processing and data transfer overheads inherent in the emulation infrastructure. The device model being emulated may or may not include timing-accurate models of the host-to-device interconnection bus, multi-device

are equivalent to the absolute and relative values reported together as the demerit figure elsewhere in the literature, except that in our case a real storage device is not always used to create the reference distribution.

(array) controller, or other non-device hardware along the request propagation path. The goal of the tuning of the emulation software is to match the response times measured at a selected measurement point—preferably a point at the outermost boundary containing all hardware included in the device model—with the internal times calculated by the device model. The most straightforward way to accomplish this is to induce the emulation software to initiate a request completion ahead of its actual completion time, such that as the completion notification propagates to the selected measurement point it arrives at precisely the desired time. Toward this end, the nature and magnitude of errors introduced by the request propagation path through the external components[5.6] must be quantified. These errors must also be quantified for the propagation path through the device drivers and bus adapter on the emulation system, as well as the errors introduced by the emulation software itself. Next, the device model must be executed in a manner that compensates for these quantified errors, while simultaneously preserving the correctness of the device model to the extent possible. Finally, the net effectiveness of this compensation must be evaluated in order to quantify the correct operation of the emulator and therefore the believability of the experimental results. Our techniques for achieving this tuning are discussed in the following subsections.

5.3.1 Execution of the device model

As presented in Section 3.3.2, the conceptual and practical execution of the timing-accurate device model during a request can be split into two interacting components. The first component, the *physical device model*, determines the time at which a request should complete. The second component, the *timing loop*, interacts with the other emulator components and with the operating system to ensure that each completion is reported at the determined time. These components can be highly integrated if desired, but our design enforces a rigid programming interface between these components—our physical device model is built upon an unmodified DiskSim codebase, and our timing loop controls the execution of DiskSim using a small set of external-interface functions—in order to enable the use of alternate physical models in the future. The necessary function prototypes for this interface in our implementation are shown in Table 5.1. There are three ways in which the physical model and the timing loop can interact, which we refer to as the run-to-completion, run-to-completion with rollback, and run-synchronously approaches.

The *run-to-completion* approach keeps the two components separate: when a request arrives, the physical device model code is invoked exactly once to determine the service time. Under this approach, requests are generally considered in isolation: the physical model uses only the real-world arrival time, the request's characteristics, and the physical model's initial internal state in its calculations. The timing loop then initiates the request completion toward the host after the determined time elapses. The emulator-based evaluation of eager writing [171] used a disk simulator by Kotz, Toh, and Radhakrishnan [105] in this manner. Although the run-to-completion approach is straightforward, it often does not properly handle concurrent requests. For example, the arrival

[5.6] Definitions of these terms describing timing-accurate storage emulation are provided in Section 3.1 (page 26).

```
Functions instantiated in the physical model:

  disksim_interface_initialize (<param file>, <callback functions>)
    function: initializes internal structures based on supplied parameters
    takes: simulator parameter file
           timing model callback functions

  disksim_interface_request_arrive (<time>)
    function: prepends a new request into the simulator's pending-event list
    takes: current simulator time, supplied by timing loop
           description of arriving request

  disksim_interface_internal_event (<time>)
    function: causes the occurance of any events scheduled before time
    takes: current simulator time, supplied by timing loop

  disksim_interface_shutdown ()
    function: cleanly shuts down disksim and prints out statistics

Callback functions instantiated in the timing loop:

  clock_schedule_callback (<time>)
    function: notifies the timing loop when to next call into physical model
    takes: invocation time of next pending event

  clock_deschedule_callback (<time>)
    function: used to unschedule an event that was reordered or removed
    takes: invocation time previously supplied to timing loop

  clock_request_complete (<time>)
    function: used to notify timing loop of request completion time
    takes: request description previously supplied by timing loop
           simulated completion time of the request
```

Table 5.1: The programming interface between the physical model and timing loop. *Although designed around an event-driven simulator under the run-synchronously approach, this interface would also support an alternative device model using the run-to-completion approach.*

of a new request may affect the service time of outstanding requests due to bus contention, request overlapping, or request scheduling.

In order to support these issues with concurrent requests, it must be possible for the physical device model to adjust its calculations for request R_1 based on the arrival times and characteristics of any request (e.g., R_2) that arrives subsequent to request R_1 but before the completion time of R_1. This can be easily accomplished using 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 ARM SEEK TO TRACK 1976 COMPLETE, READ OF SECTORS 620464–620492 INTO READ BUFFER 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 the simulator's internal device state and schedules subsequent events. For example, when one of the READ OF SECTORS INTO READ BUFFER COMPLETE events (event e) occurs for read request R_1, the simulator may decide to schedule a set of events representing the bus acquisition and transfer of the buffered blocks for R_1. However, if write request R_2 arrived in the simulator just prior to event e, the simulator may instead decide to first schedule the bus transfer of the blocks for R_2 into the disk's write buffer before scheduling the acquisition and bus transfer of the data for R_1.

Under the *run-to-completion with rollback* approach, the physical device model software is speculatively executed to request completion upon receipt of each request (e.g., R_1) by the physical model, with this calculated completion time reported immediately to the timing loop. Whenever a subsequent request R_2 arrives before the actual completion time of R_1, the physical model is notified of this and instructed to re-calculates the completion time for R_1 using the additional knowledge about R_2. Practically, this requires that the physical model support either mid-request checkpointing or rollback to a specified time before the request completion. In the example of the previous paragraph, the expected response time ΔT_1 for R_1 would be calculated by the physical model immediately upon the arrival of R_1. Later, when R_2 arrived before the completion time of R_1, the physical model state would be rolled back (or a checkpoint restored) in order for a new $\Delta T_1'$ to be calculated taking into account knowledge of both requests. The mechanism for this "rewinding" of the device state could either aim to revert back to the arrival time of R_1 or only back to the arrival of R_2—depending on the implementation of the physical model, this may exercise an interesting trade-off between the processing resources required to repeatedly execute the physical model's calculations for a single request (the former approach), versus the resources required to support a fine-grained event- or time-based checkpoint/rollback facility (the latter approach). Additionally, there may be a brief time immediately before the end of R_1 where it is not computationally possible to consider any aspect of R_2 without unduly delaying the completion of R_1. In practice we were unable to pursue this approach because of difficulties in extending the DiskSim simulator to efficiently support fine-grained checkpointing.

The *run-synchronously* approach eliminates the need for checkpointing or rollback support in the physical device model. Under this approach, which is the approach we follow in our implementation, the advancement of the simulator's internal clock is synchronized with the advancement of the real-world ("wall") clock. When each event occurs inside the physical model, it informs the timing loop of the invocation time of the next event it schedules. In turn, the timing loop calls back into the physical model when the wall clock time matches the invocation time of the next event. When the time of the REQUEST COMPLETE event is ultimately scheduled inside the physical model, the timing loop is specially notified and initiates the request completion toward the host once that time arrives. One concern with this approach is that some request completions may not occur at the scheduled time because of the delayed processing of the request; in the earlier approaches all processing was completed immediately, whereas in this approach the processing is distributed over the lifetime of a request, including near the request completion time. This problem may be exacerbated if many internal events, such as simulated buffer interaction events or bus transfer events, are unevenly distributed to occur near the end of a request. To avoid this, the physical model can be speculatively executed a short time into the future—i.e., the timing loop can call back into the physical model slightly before the actual wall clock time that matches the next event—such that the final notification of a request's completion time is always presented to the timing loop slightly in advance.

5.3.2 Error-cognizant execution of the physical device model

Once an understanding has been developed concerning the nature of the per-request errors and the magnitude of $E_{1\rightarrow M}$ is known, the physical device model may be executed in such a way that request completion propagations arrive at the host system at the desired time. Specifically, the measured arrival time and calculated completion times may be skewed slightly inside the emulator to account for the request processing and communication delays, with the goal of equalizing the elapsed time at MP_1 and MP_3. In our implementation, this means making sure the elapsed request service time measured in the host system matches the elapsed service time computed by the device simulation model.

To accomplish this correction, we consider individually the two constituent components of $E_{1\rightarrow M}$: the arrival-time error ($E^A_{1\rightarrow M}$) and the completion-time error ($E^C_{1\rightarrow M}$). These terms are defined byEquation 5.9 and their relationship to the experimental error is illustrated in Figure 5.3.

$$E_{1\rightarrow M} = E^A_{1\rightarrow M} + E^C_{1\rightarrow M} \tag{5.9}$$

As it is nontrivial to accurately and precisely synchronize the clocks between the host and emulation systems [107], we assume in our implementation (a remote emulator where $M = 3$) that these values are equal, as defined by Equation 5.10. For the remainder of this dissertation we use the term ΔT_{skew} to represent our experimental configuration of these values.

$$E^{A}_{1\to 3} = E^{C}_{1\to 3} = \frac{E_{1\to 3}}{2} \equiv \Delta T_{skew} \tag{5.10}$$

The mitigation of these constituent errors is discussed in the following two sections. These discussions make use of the terminology presented in Table 5.2.

5.3.2.1 Compensating for request arrival-time errors

Request arrival-time propagation delays can be accounted for in the emulation software by artificially reducing the arrival time of requests as they are presented to the physical device model. Under this *arrival-time modification* technique, illustrated in Figure 5.4, requests are inserted into the physical model with a modified $T_{arrive} = T_{current} - E^{A}_{1\to M}$. The goal of this technique is twofold: first to ensure the accurate response time is measured inside the physical model (MP_3 in our implementation), and second to ensure the physical model has the most accurate information about its internal state and the state of other requests at the time the request arrives.

Arrival-time modification will generally not affect the computed completion times of the requests, except that it may cause the requests to (correctly) complete more quickly. Consider a greatly simplified scenario where a read request for block 714 arrives at an emulated disk. Assuming the emulated disk arm is already positioned over the correct track, then if the request arrives at wall-clock time T_{713} (which in this example conveniently corresponds with the time the emulated disk head begins passing over block 713) then the physical model will complete the request very quickly, after the disk head passes over blocks 713 and 714. However, if the arrival delay is large—for example, if $E^{A}_{1\to M} = 5$ and the new request does not therefore arrive at the emulator until time T_{718}—then the request will take far longer to complete, as almost a full rotation of the disk platter will occur before block 714 is again available for reading. Arrival-time modification prevents this problem. Unfortunately, the technique cannot always be used due to limitations in the run-synchronously approach, as explained in Section 5.3.3.

5.3.2.2 Compensating for request completion-time errors

Request completion-time propagation delays can be accounted for in an emulator by instructing the emulation software to transmit the request completion early to the host. Under this *early-completion* technique, illustrated in Figure 5.5, the request completion is sent by the communications manager when $T_{current} = T_{complete} - E^{C}_{1\to M}$ to fully account for the request processing and communication delays. This includes the delays inherent to process scheduling and OS-level interactions with the emulation software. The early-completion technique can be used in isolation or can be combined with the arrival-time modification technique to effect the best overall error compensation time.

Use of the early-completion technique is straightforward under the run-to-completion and run-to-completion with rollback approaches. Under the run-synchronously approach used in our implementation, the simtime (the clock in the physical device model) must be maintained at least $E^{C}_{1\to 3}$

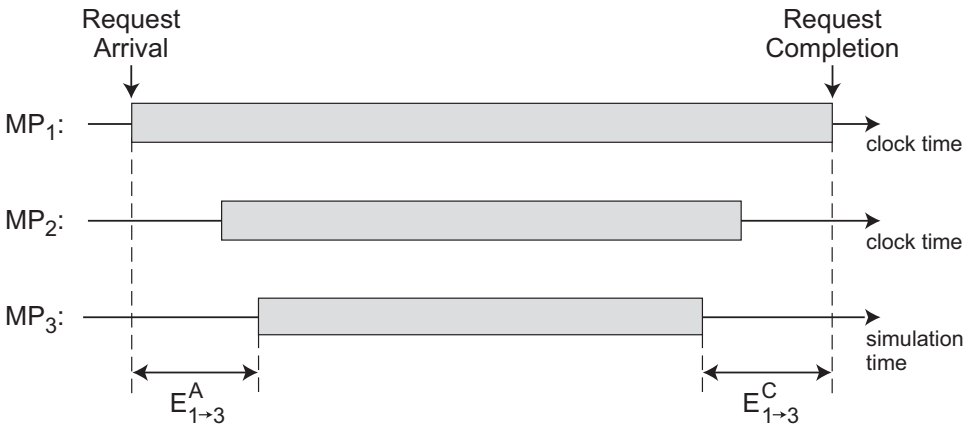


Figure 5.3: Request service times at the various measurement points. *In this example, measurement point MP_1 is in the host system, MP_2 is in the emulation system, and MP_3 is inside the physical device model. The shaded area represents the interval between the request arrival and request completion at each measurement point. The objective of error compensation is to equalize the times measured at MP_1 and MP_3.*

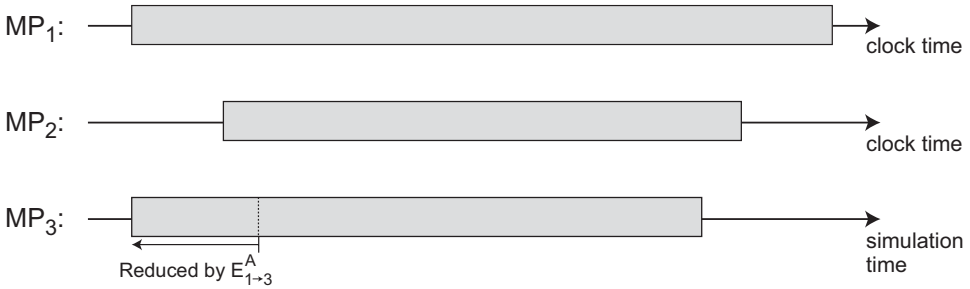


Figure 5.4: Compensating for request arrival-time errors. *As explained in Section 5.3.2.1, the request arrival time presented to the physical device model may be reduced $E^{A}_{1\rightarrow M}$ time units into the past. This compensates for the propagation delays from the host to the emulation software.*

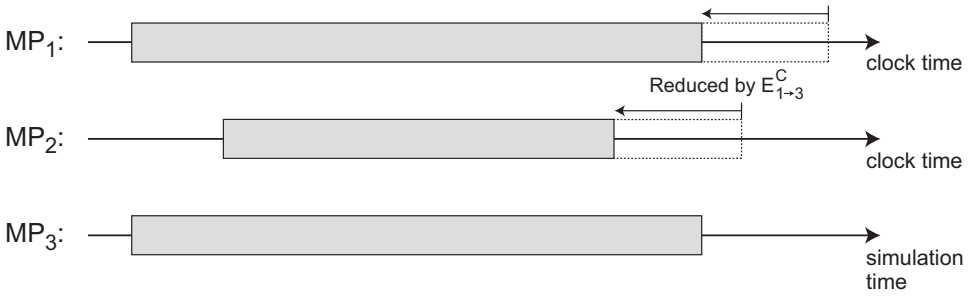


Figure 5.5: Compensating for request completion-time errors. *As explained in Section 5.3.2.2, the simulation time may be advanced into the future (prematurely but intentionally) to determine a request's completion time in advance of the actual time. This allows the communications manager to transmit the completion $E^{C}_{1\rightarrow M}$ time units early—compensating for the propagation delays from the emulation software to the host—which will result in the completion notification arriving at the correct time in the host system. (In the previous figures, the notification arrives late at MP_1.)*

	Absolute time specifiers
$T_{current}$	The current real-world (wall-clock) time maintained by the timing loop.
T_{arrive}	The simulated arrival time of a request as transmitted from the emulator to the physical model. Note that use of the arrival-time modification technique causes T_{arrive} to not necessarily equal $T_{current}$ at the time the request arrived inside the emulator.
$T_{complete}$	The simulated completion time of a request as calculated by the physical model and transmitted to the communications manager.
T_{event}	The simulated time of the most recent simulation event E that occurred inside the physical model. Note that the use of the early completion technique (or $\Delta T_{lookahead}$) causes T_{event} to not necessarily have equaled $T_{current}$ when E was processed.
	Relative time intervals
$E^{A}_{1\rightarrow M}$	The component of $E_{1\rightarrow M}$ caused during the propagation of the request arrival notification from the host system to the emulation software. This is also the degree to which the arrival time is reduced under the arrival-time modification technique.
$E^{C}_{1\rightarrow M}$	The component of $E_{1\rightarrow M}$ caused during the propagation of the request completion notification from the emulation software to the host system. This is also the degree to which the completion notification is sent early by the communications manager under the early-completion technique.
ΔT_{skew}	A configuration parameter for our timing-accurate storage emulator implementation. During experimentation, the values of $E^{A}_{1\rightarrow M}$ and $E^{C}_{1\rightarrow M}$ are initialized to equal this value.
$\Delta T_{lookahead}$	An extra interval (beyond $E^{C}_{1\rightarrow M}$) by which the clock in the physical device model is kept ahead of $T_{current}$ during the execution of a request, when using the early-completion technique under the run-synchronously approach.

Table 5.2: Nomenclature for the discussion of techniques for mitigating the per-request error. *This terminology supports the discussion in Section 5.3.2.*

time units ahead of the wall-clock time during the execution of request in order for the communications manager to be able to send the completion early. In practice, we maintain the simtime ahead of the wall-clock time by a slightly larger value ($E^C_{1\rightarrow 3} + \Delta T_{lookahead}$).[5.7] We refer to this extra interval as the *additional lookahead*. It is not necessary for an emulator to support the use of this additional lookahead; however, as noted in Section 5.4.1, use of $\Delta T_{lookahead}$ has the advantage of eliminating any variable completion-time processing delays that would otherwise increase $E_{2\rightarrow 3}$.

5.3.3 Limitations of error compensation

One limitation when considering error compensation is that none of the three approaches from Section 5.3.1 will be able to accurately satisfy any request with a response time $\Delta T < E_{1\rightarrow M}$. The implications of this are that our implementation is unable to service requests shorter than 0.12 ms using the Adaptec SCSI adapter or 0.30 ms using the QLogic Fibre Channel adapter. In such a case, where the round-trip delay introduced between the measurement point and the emulator is greater than the response time itself, one possible workaround is to "short-circuit" the request path by placing special software in the device driver or operating system on either the host or emulation system to detect such a request and service it immediately from that location—in effect, creating a hybrid remote/local emulation system. This may be practical when using a workloads known to be deterministic, or workloads where it is possible to predict when low response time requests would occur based on the characteristics of the request stream leading up to such a request. As discussed before, it may also be possible to balance the response time overage of such a request by servicing subsequent requests slightly faster than the times reported by the physical model.

A similar concern is that the total processing time for a request must be less than ΔT (or $\Delta T - E^C_{1\rightarrow M}$ when using the early-completion technique). Our initial attempts at emulation of a MEMS-based storage device model were thwarted by the excessive computational requirements of the MEMS device simulator.[5.8] When request deadlines simply cannot be met by the emulation software, a similar workaround of utilizing special interception software in the host or emulation system may be possible.

Although our initial experimentation with the Memulator attempted to use the arrival-time reduction technique, we were unable to fully make use of this reduction in our current implementation. This is because we have been unable to validate the behavior of the DiskSim simulator when it is presented with simulation times that are not monotonically nondecreasing. As demonstrated in Figure 5.6, it is nontrivial to use the arrival-time reduction technique when using the physical model

[5.7] Note that we still do not transmit the request completion until $T_{current} = T_{complete} - E^C_{1\rightarrow M}$; the use of $\Delta T_{lookahead}$ is intended to provide a small time buffer before the completion is sent, with the goal of reducing the variance of the effectiveness of error compensation.

[5.8] Solving this problem of meeting the real-time requirements of emulation ultimately required us to re-architect the MEMS device simulator to enable the internal caching of many of its intermediate computational results. This led to an interesting philosophical debate concerning the unsolved problem of balancing the desire for "fast development of working and tested code" versus the "worthwhile pursuit of obvious opportunities for efficiency in system software design." We leave the solution to this debate as an exercise for the reader.

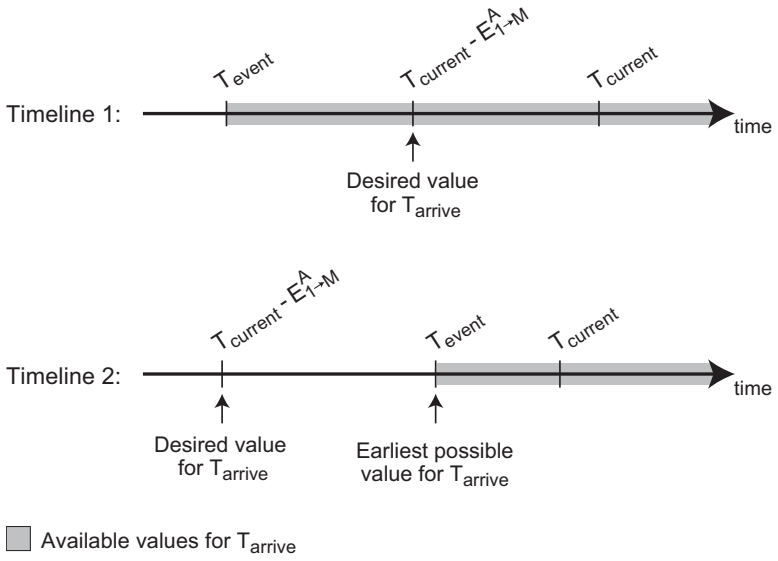


Figure 5.6: A concern with the arrival-time reduction technique when accounting for request propagation delays under the run-synchronously approach. *The terms in this figure are introduced in Table 5.2. When $T_{event} < (T_{current} - E^{A}_{1\rightarrow M})$, as shown in Timeline 1, it is possible to reduce the arrival time by the full value of $E^{A}_{1\rightarrow M}$. However, when $(T_{current} - E^{A}_{1\rightarrow M}) < T_{event}$, as shown in Timeline 2, the arrival time cannot be reduced by the full value without reversing the flow of time as presented to the physical device model. This can be nontrivial; the evaluator should first validate the correct operation of the physical device model under such nonintuitive circumstances.*

under a run-synchronously approach. If any previous event for another request occurred within ΔT_{reduce} of the current time (i.e., $T_{current} - E^{A}_{1\rightarrow M} < T_{event}$) then T_{arrive} cannot be fully reduced to the desired value without presenting an out-of-order time to the physical model. This problem of overlapping times will be less of a concern when using one of the run-to-completion approaches for executing the physical model.

This problem of out-of-order values for T_{arrive} and T_{event} is exacerbated when using the early-completion technique under the run-synchronously approach. Because T_{event} is being artificially skewed into the future, it is less likely that use of the arrival-time reduction techniques will be possible for a request that arrives shortly after ($< \Delta T_{reduce}$) the completion of the previous request. Moreover, this technique introduces the problem that it may not even be possible to set $T_{arrive} = T_{current}$; in our implementation, we calculate arrival time based on Equation 5.11:[5.9]

$$T_{arrive} = \begin{cases} T_{event} & \text{if } (T_{event} > T_{current}) \\ T_{current} & \text{otherwise} \end{cases} \tag{5.11}$$

Because we are unable to use arrival-time modification, some requests will incur an extra rotation of the disk platter before completing. In the worst case this will cause an occasional request to incur an

[5.9] Equation 5.11 indicates that we do not insert the request into our device simulator with a reduced arrival time. However, we still calculate the request service time at MP_3 using a fully reduced arrival time: $T_{arrive} = T_{current} - E^{A}_{1\rightarrow 3}$.

unnecessary full rotational delay before completion, causing the emulator to slightly under-perform with respect to the true disk model performance. We note that for the disk model used in the calibration experiments in Section 5.4, values of 60 μs and 151 μs for $E_{1\rightarrow3}$ were obtained, which respectively represent 1% and 2.5% of the 6 ms platter rotation time for the disk model.

Additional workload analysis can be used to determine the degree to which these unmodified arrival-time requests are a problem. To achieve this, the experimental workload can be traced inside the emulator, storing $T_{current}$ at the time of each request's arrival at the physical model as well as the calculated value of $T_{complete}$. This trace can then be replayed post-experimentally against the stand-alone physical model using the principles of the arrival-time modification technique, to determine whether any different values for $T_{complete}$ should have been calculated. Alternatively, this could be checked while an experiment is in progress by running multiple simultaneous instantiations of the physical model: a primary instantiation connected to the timing loop using the run-synchronously approach, and a secondary instantiation driven under the run-to-completion approach with arrival-time modification—with an internal clock lagging at least ΔT behind the primary—used to check whether the times reported by the primary are indeed accurate.

5.4 Calibrating the emulation software for the reduction of response time errors

This section presents the empirical methodology we used to quantify the errors introduced by the request arrival and completion propagation paths. The errors we quantify—$E_{2\rightarrow3}$ represents the error between the emulation software and the physical device model times, and $E_{1\rightarrow2}$ represents the error between the measurement points in the host system and the emulator software—are discussed above in Section 5.2.2.

Our evaluation uses a series of trace-driven microbenchmarks that repeatedly exercise a single characteristic request. For example, each of these traces contain a fixed-length read from a fixed offset, repeated many times in succession with a fixed interarrival time. Using single-characteristic traces allows multiple experimental runs to be compared to determine whether the errors are dependent on any readily-available request characteristics, such as the type of the request or the request length.

Our mechanism for repeatedly exercising a characteristic request was the trace-replay utility (`dxreplay`) of the Dixtrac disk characterization toolkit developed by Jiri Schindler, John Bucy, and Greg Ganger. This is a user-level utility that reads in a trace file consisting of request arrival times, the block offset and block length for each request, and a flag representing the request type (read or write). Each request is sent to the device using the Linux SCSI Generic (sg) bus interface at the appropriate time as specified in the trace file. These experiments use data collected from replaying eight traces. The traces represent the cross-product of four request sizes (1 KB, 4 KB, 32 KB, and 64 KB) with the two request types. Each trace contains 1000 requests with a 50 ms interarrival time between requests. As only one request is sent to the emulated device at a time, this interarrival time ensures each request completes before the following request is initiated. The emulated device model

used for these experiments is a validated model for the Seagate Cheetah disk described in Table 4.1 on page 44, which is also the Emulated Cheetah model used for the experiments in Chapter 6.

The remainder of this section is organized as follows.

The figures shown in this section contain four fixed-axis subgraphs: (1) the individual request errors [upper left], (2) a histogram of the individual request errors [upper right], (3) a histogram of the individual request percentage errors [lower left], and (4) the aggregate request error distribution [lower right]. The quantitative metrics for the experimental run are included in the title of each subgraph. The domain and range are fixed in each subgraph for easier comparison among the figures. Fixing the axes in this way caused abridgment of some of the extreme outliers on the graphs, so full graphs of the individual request errors for each experiment are provided in Appendix D. To create these figures, each trace was experimentally executed three times, and the subgraphs and quantitative metrics presented in each figure represent the best or lowest values obtained across the three experiments.[5.10]

5.4.1 Error between the emulation software and the physical device model

Our analysis of the factors contributing to $E_{2\rightarrow3}$ begins with the trace of repeated eight-block reads using the Adaptec bus adapter. The initial results for this trace are shown in Figure 5.7.

The results indicate a dependency between the values calculated for $E_{2\rightarrow3}$ and $\Delta T_{lookahead}$, one of the values used for the early-completion lookahead technique in the timing loop. To explore this we repeated the eight-block Adaptec read trace using values for $\Delta T_{lookahead}$ over the range [0 μs, 60 μs] in increments of 1 μs. The resulting data are shown in Figure 5.8. These data indicate that for values of $\Delta T_{lookahead} < 28\,\mu$s there is an additional delay present in $E_{2\rightarrow3}$ that inversely correlates with $\Delta T_{lookahead}$. This suggests that there are two components of $E_{2\rightarrow3}$: one fixed component of 18 μs that represents the emulation software overheads (in the timing loop and communications manager), and a variable component caused by incomplete calculations in the physical device model as the request nears completion. When operating with $\Delta T_{lookahead}$ values greater than that shown on the inflection point—which we do due to the large $E_{1\rightarrow2}$ calculated below—the experimenter should calculate $E_{2\rightarrow3}$ with the lookahead enabled. The remainder of the experiments in this section use a lookahead value of 30 μs.

Using this lookahead value yields the corrected data for the eight-block Adaptec read trace shown in Figure 5.9. This figure is used as the standard against which the remaining figures for the Adaptec driver in this section are compared. The full results for the experiments in this section,

[5.10] Using the best or lowest values here (instead of mean values, for example) is appropriate for calibration purposes because the emulator should be calibrated with the best available data—i.e., the data containing the fewest large errors.

representing all eight traces for both the Adaptec and QLogic bus adapters, are presented in Table 5.3 and Table 5.4. As with the figures, the data in these tables represent a composite of the best values obtained after executing each trace three times.

With our current implementation of the Memulator, we observe no dependence for $E_{2\rightarrow 3}$ on the size of the request.[5.11] This is evident in a visual comparison of the similarity of the data between the 128-block Adaptec read trace with lookahead, shown in Figure 5.10, and the data for the eight-block Adaptec read trace with lookahead in Figure 5.9. This is also evident by direct inspection of the values in Table 5.3.

In our current implementation, we observe almost no dependence (less than 5 μs) for $E_{2\rightarrow 3}$ on the type of the request. This is evident in a visual comparison of the similarity of the data between the eight-block Adaptec write trace with lookahead, shown in Figure 5.11, and the data for the eight-block Adaptec read trace with lookahead in Figure 5.9. This is also evident by direct inspection of the values in Table 5.3.

As expected, we observe no dependence for $E_{2\rightarrow 3}$ on the choice of host-to-target interconnect. This is evident in a visual comparison of the similarity of the data between the eight-block QLogic read trace with lookahead, shown in Figure 5.14, and the data for the eight-block Adaptec read trace in Figure 5.9. This is also evident by direct inspection of the values in Table 5.3 and Table 5.4.

We additionally confirm each of the earlier findings based on the Adaptec driver by repeating each of the trace-based experiments using the QLogic bus adapter. After determining the baseline performance under no lookahead via the trace of repeated eight-block reads (Figure 5.12), we confirm the dependence on the additional lookahead (Figure 5.13) and the independence on request size (Figure 5.15) and request type (Figure 5.16). These findings are also evident by direct inspection of the values in Table 5.4.

In summary, $E_{2\rightarrow 3}$ exhibits a dependence on the additional lookahead value $\Delta T_{lookahead}$, and exhibits no dependence on the request size or type, or the choice of interconnect. The quantified value for our implementation is shown in Equation 5.12.

$$E_{2\rightarrow 3} = \begin{cases} 46 - \Delta T_{lookahead}\ \mu\text{s} & \text{if } (\Delta T_{lookahead} < 28\ \mu\text{s}) \\ 18\ \mu\text{s} & \text{otherwise} \end{cases} \tag{5.12}$$

[5.11] As discussed in Section 5.2.2, inefficiencies in our previous implementation resulted in dependencies for $E_{2\rightarrow 3}$ on both request size and type.

$\Delta T_C \rightarrow \Delta T_R$ (mem→sim)	$\overline{\Delta T_R}$ [ms]	EI_{avg} [ms]	$EI_{avg\%}$ [%]	EI_{rms} [ms]	$EI_{rms\%}$ [%]	EA_{rms} [ms]	$EA_{rms\%}$ [%]
Adaptec							
2 block read	3.077	0.017	0.55	0.020	0.64	0.018	0.60
2 block write	3.148	0.020	0.63	0.021	0.66	0.020	0.64
8 block read	3.172	0.017	0.52	0.017	0.55	0.017	0.53
8 block write	3.235	0.020	0.61	0.020	0.63	0.020	0.61
64 block read	3.894	0.017	0.43	0.017	0.44	0.017	0.43
64 block write	3.999	0.023	0.57	0.043	1.08	0.023	0.58
128 block read	4.704	0.018	0.38	0.019	0.40	0.018	0.38
128 block write	4.792	0.023	0.47	0.024	0.50	0.023	0.48

Table 5.3: $E_{2\rightarrow 3}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=0** μ**s, Adaptec.**

$\Delta T_C \rightarrow \Delta T_R$ (mem→sim)	$\overline{\Delta T_R}$ [ms]	EI_{avg} [ms]	$EI_{avg\%}$ [%]	EI_{rms} [ms]	$EI_{rms\%}$ [%]	EA_{rms} [ms]	$EA_{rms\%}$ [%]
Qlogic							
2 block read	3.069	0.020	0.65	0.025	0.83	0.025	0.80
2 block write	3.206	0.023	0.71	0.024	0.76	0.024	0.75
8 block read	3.156	0.023	0.72	0.079	2.52	0.040	1.25
8 block write	3.288	0.027	0.81	0.081	2.46	0.028	0.86
64 block read	3.870	0.022	0.58	0.078	2.01	0.024	0.61
64 block write	3.970	0.029	0.73	0.096	2.42	0.066	1.67
128 block read	4.713	0.022	0.47	0.086	1.82	0.066	1.41
128 block write	4.799	0.028	0.57	0.075	1.56	0.059	1.23

Table 5.4: $E_{2\rightarrow 3}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=0** μ**s, QLogic.**

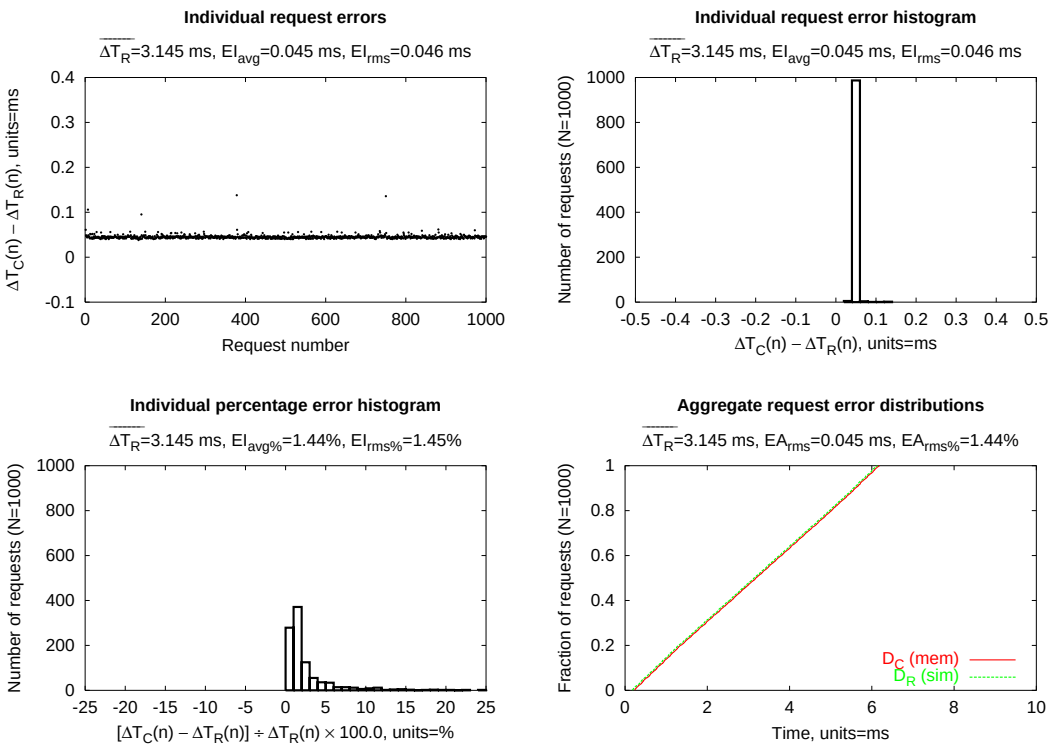


Figure 5.7: $E_{2\rightarrow3}$**:** $\Delta T_{lookahead}$**=0** μ**s,** ΔT_{skew}**=0** μ**s, Adaptec, 4 KB Reads.**

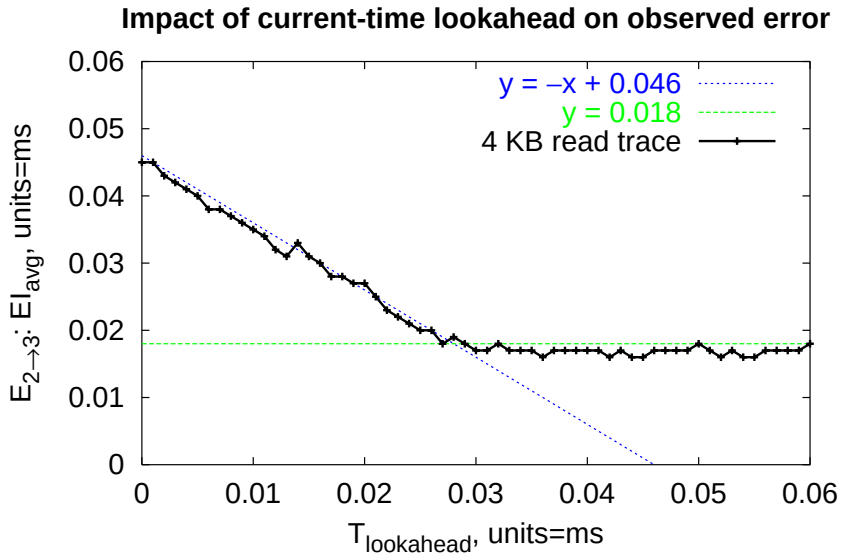


Figure 5.8: The effect of $\Delta T_{lookahead}$ **on** $E_{2\rightarrow3}$**, Adaptec.**

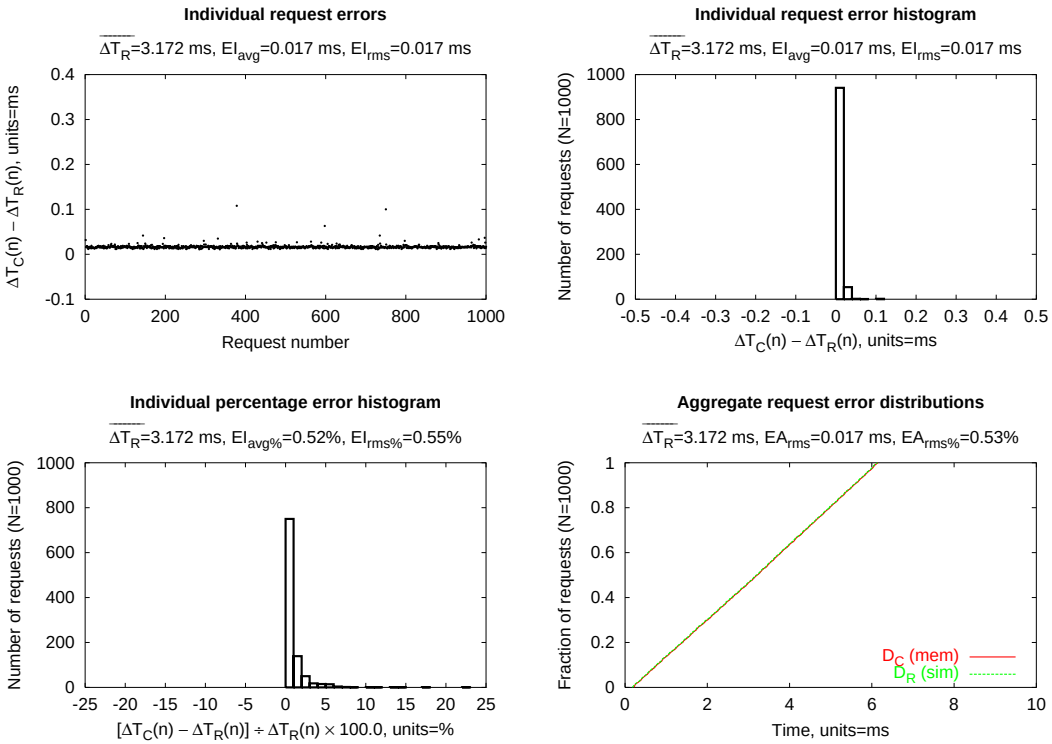


Figure 5.9: $E_{2\rightarrow3}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=0** μ**s, Adaptec, 4 KB Reads.**

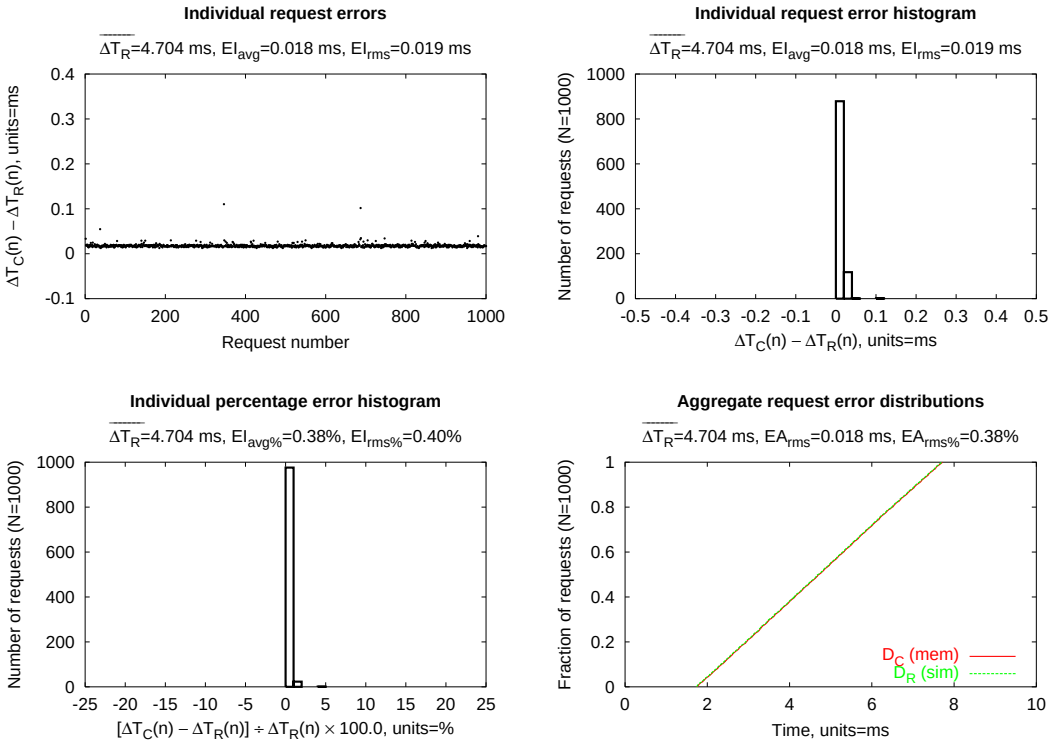


Figure 5.10: $E_{2\rightarrow3}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=0** μ**s, Adaptec, 64 KB Reads.**

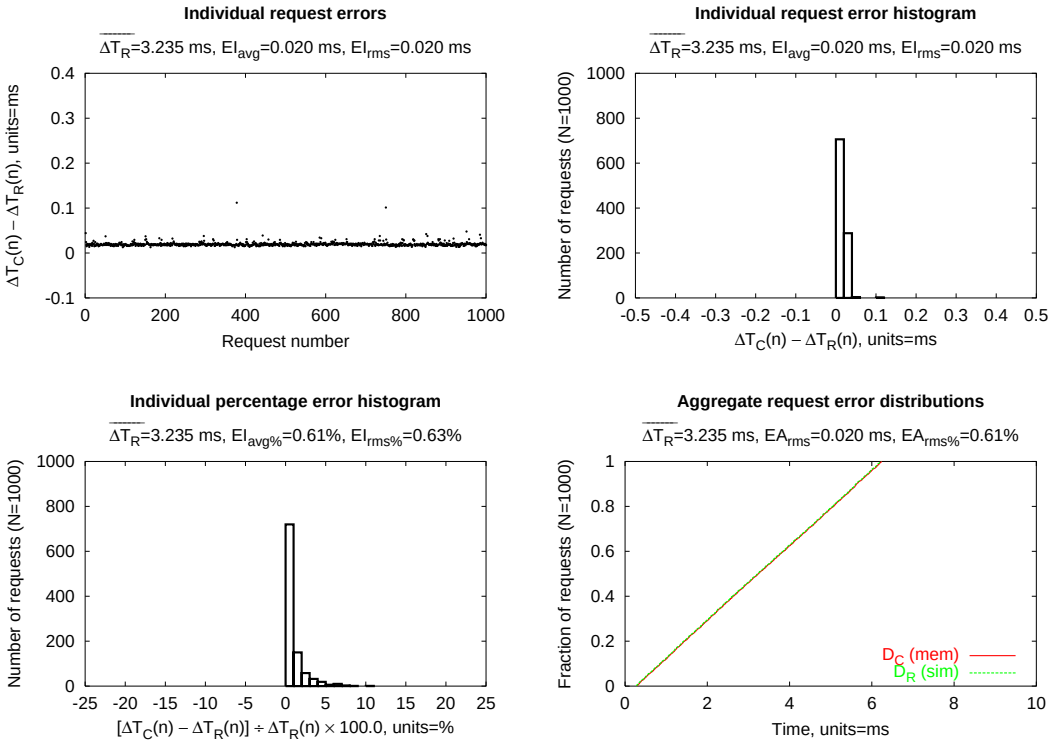


Figure 5.11: $E_{2\rightarrow3}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=0** μ**s, Adaptec, 4 KB Writes.**

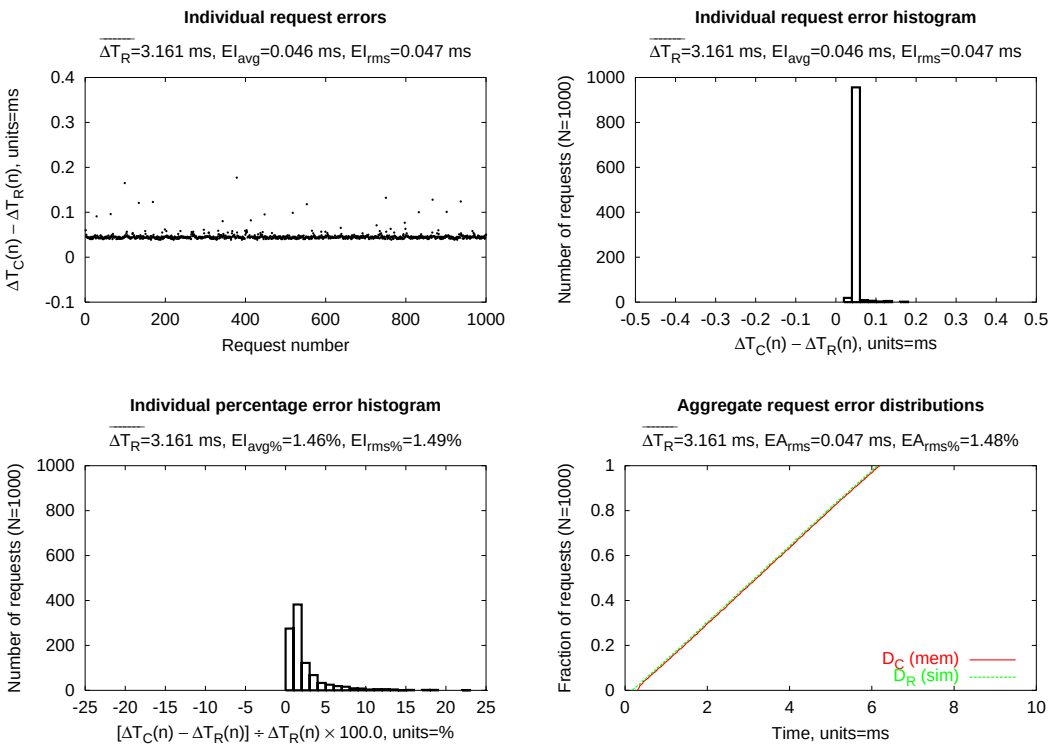


Figure 5.12: $E_{2\rightarrow3}$**:** $\Delta T_{lookahead}$**=0** μ**s,** ΔT_{skew}**=0** μ**s, QLogic, 4 KB Reads.**

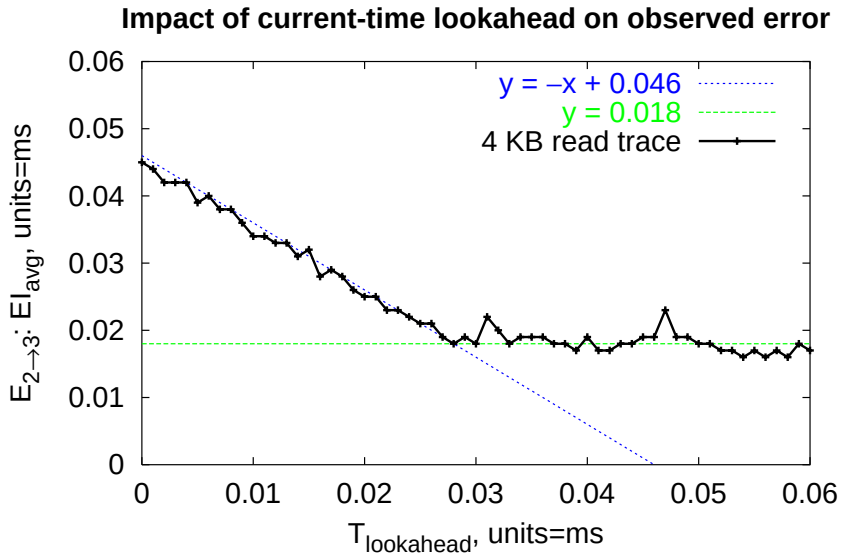


Figure 5.13: The effect of $\Delta T_{lookahead}$ **on** $E_{2\rightarrow3}$**, QLogic.**

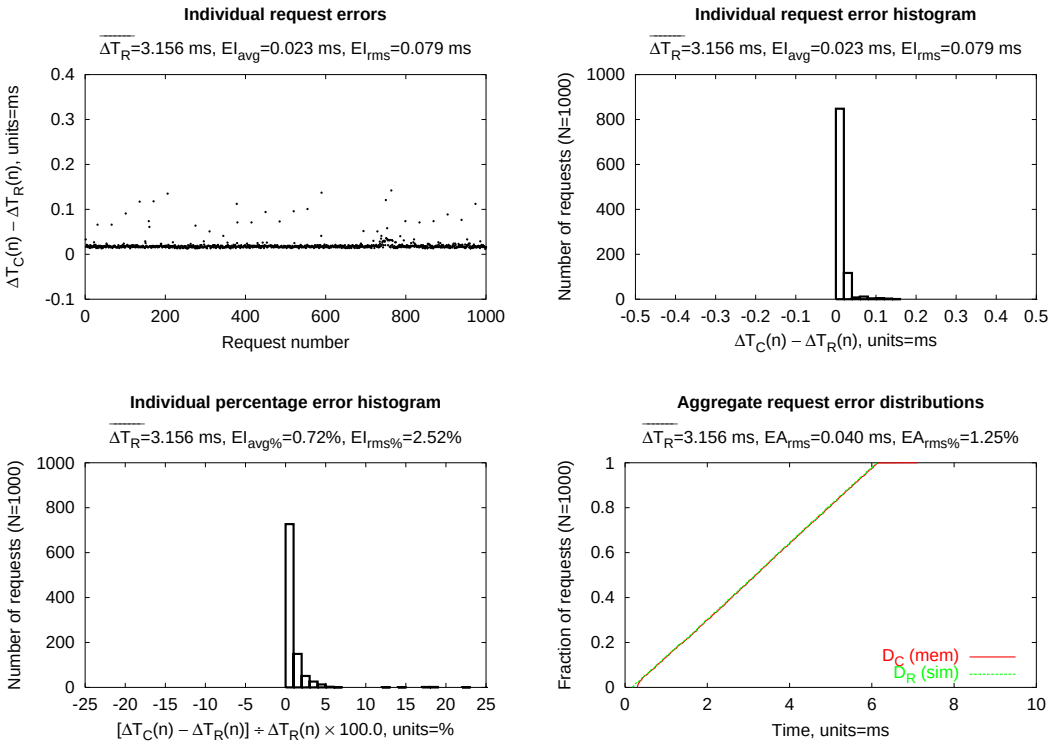


Figure 5.14: $E_{2\rightarrow3}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=0** μ**s, QLogic, 4 KB Reads.**

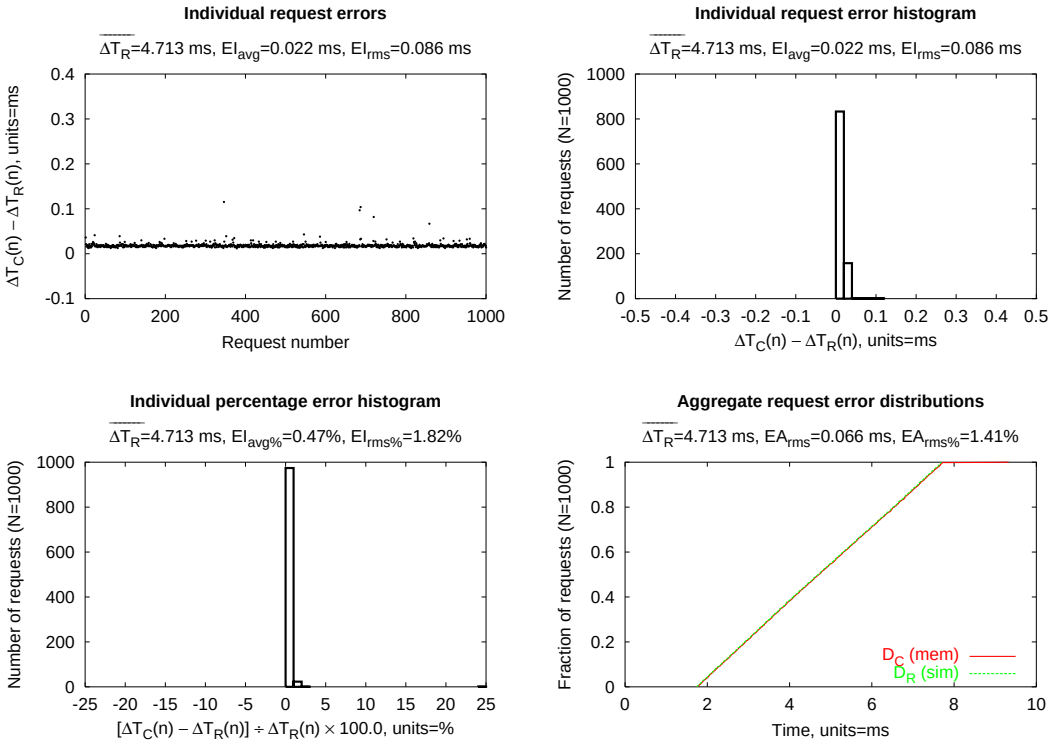


Figure 5.15: $E_{2\rightarrow3}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=0** μ**s, QLogic, 64 KB Reads.**

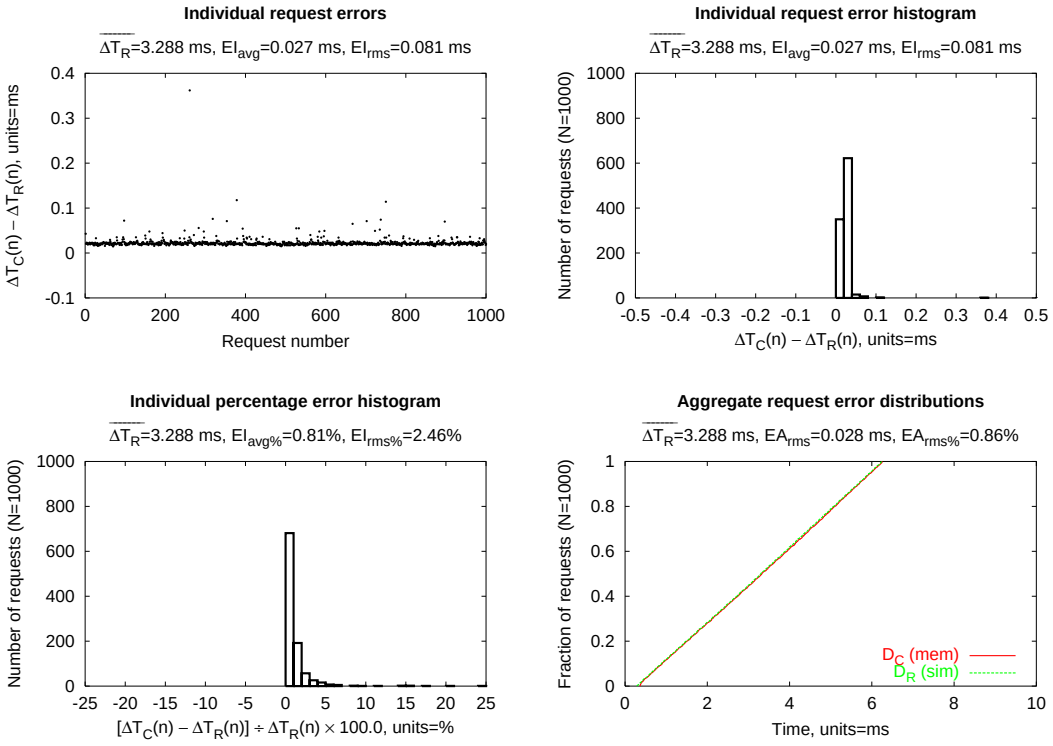


Figure 5.16: $E_{2\rightarrow3}$: $\Delta T_{lookahead}$=30 μs, ΔT_{skew}=0 μs, QLogic, 4 KB Writes.

5.4.2 Error between the host and the emulation software

Following the pattern established in the previous section, the analysis of the factors contributing to $E_{1\rightarrow2}$ begins with the trace of the repeated eight-block reads using the Adaptec bus adapter. The results for this trace are shown in Figure 5.17. Additionally, the full results for the experiments in this section, representing all eight traces for both the Adaptec and QLogic bus adapters, are presented in Table 5.5 and Table 5.6. As with the figures, the data in these tables represent a composite of the best values after executing each trace three times.

Unlike the dependency observed between the values calculated for $E_{2\rightarrow3}$ and $\Delta T_{lookahead}$, we discern no dependency between the values calculated for $E_{1\rightarrow2}$ and the value used for the additional lookahead. To explore this we repeated the eight-block Adaptec read trace using values for $\Delta T_{lookahead}$ over the range [0 μs, 400 μs] in increments of 10 μs. The resulting data are shown in Figure 5.18, and demonstrate no variation in the calculated value of $E_{1\rightarrow2}$ across the range of values for $\Delta T_{lookahead}$. These results are as expected; the use of $\Delta T_{lookahead}$ mitigates the variable-time overheads of the emulation software, which should have no effect on the propagation delays across the storage interconnect. The remainder of the experiments in this section use a lookahead value of 30 μs to match the value used previously.

As with $E_{2\rightarrow3}$, we observe no dependence for $E_{1\rightarrow2}$ on the size of the request. This is evident in a visual comparison of the similarity of the data between the 128-block Adaptec read trace, shown in Figure 5.20, and the data for the eight-block Adaptec read trace in Figure 5.19. This is also evident by direct inspection of the values in Table 5.5. This independence is as expected; any delays caused by excessive request transfer time—such as when very large requests are being transferred and the scope of the emulator includes a fast bus—will appear as delays in the communications manager, and will result in an increased $E_{2\rightarrow3}$. The delays represented by $E_{1\rightarrow2}$ are wholly in the transmission of the fixed-size request arrival and completion notifications.

Also as with $E_{2\rightarrow3}$, we also observe no dependence for $E_{1\rightarrow2}$ on the type of the request. This is evident in a visual comparison of the similarity of the data between the eight-block Adaptec write trace, shown in Figure 5.21, and the data for the eight-block Adaptec read trace in Figure 5.19. This is also evident by direct inspection of the values in Table 5.5.

Unlike $E_{2\rightarrow3}$, we do observe a dependence for $E_{1\rightarrow2}$ on the choice of host-to-target interconnect. The error when using the QLogic driver is almost three times as large (282 μs) than the error when using the Adaptec driver (99 μs). This disparity is evident in a visual comparison of the dissimilarity of the data between the eight-block QLogic read trace, shown in Figure 5.24, and the data for the eight-block Adaptec read trace in Figure 5.19. This is also evident by direct inspection of the values in Table 5.5 and Table 5.6.

We additionally confirm each of the earlier findings based on the Adaptec driver by repeating each of the trace-based experiments using the QLogic bus adapter. After determining the baseline performance under no lookahead via the trace of repeated eight-block reads (Figure 5.24), we confirm the independence from the current-time lookahead (Figure 5.23), request size (Figure 5.25)

and request type (Figure 5.26). These findings are also evident by direct inspection of the values in Table 5.6.

In summary, $E_{1\to 2}$ exhibits a dependence on the choice of interconnect, and exhibits no dependence on the current-time lookahead value or the request size or type. The quantified value for our implementation is shown in Equation 5.13.

$$E_{1\to 2} = \begin{cases} 99\ \mu\text{s} & \text{for the Adaptec bus adapter} \\ 282\ \mu\text{s} & \text{for the QLogic bus adapter} \end{cases} \tag{5.13}$$

$\Delta T_C \rightarrow \Delta T_R$ (hos→mem)	$\overline{\Delta T_R}$ [ms]	EI_{avg} [ms]	$EI_{avg\%}$ [%]	EI_{rms} [ms]	$EI_{rms\%}$ [%]	EA_{rms} [ms]	$EA_{rms\%}$ [%]
Adaptec							
2 block read	3.094	0.097	3.12	0.097	3.13	0.097	3.13
2 block write	3.186	0.097	3.04	0.097	3.04	0.097	3.04
8 block read	3.162	0.097	3.08	0.098	3.09	0.098	3.08
8 block write	3.291	0.097	2.95	0.097	2.95	0.097	2.95
64 block read	3.910	0.097	2.48	0.097	2.49	0.097	2.49
64 block write	4.026	0.098	2.42	0.098	2.43	0.098	2.43
128 block read	4.751	0.099	2.08	0.099	2.09	0.099	2.08
128 block write	4.814	0.098	2.04	0.098	2.04	0.098	2.04

Table 5.5: $E_{1\rightarrow 2}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=0** μ**s, Adaptec.**

$\Delta T_C \rightarrow \Delta T_R$ (hos→mem)	$\overline{\Delta T_R}$ [ms]	EI_{avg} [ms]	$EI_{avg\%}$ [%]	EI_{rms} [ms]	$EI_{rms\%}$ [%]	EA_{rms} [ms]	$EA_{rms\%}$ [%]
Qlogic							
2 block read	3.090	0.282	9.13	0.282	9.13	0.282	9.13
2 block write	3.238	0.281	8.68	0.281	8.68	0.281	8.68
8 block read	3.164	0.282	8.91	0.282	8.91	0.282	8.91
8 block write	3.268	0.281	8.59	0.281	8.59	0.281	8.59
64 block read	3.938	0.281	7.14	0.281	7.14	0.281	7.14
64 block write	3.998	0.281	7.04	0.281	7.04	0.281	7.04
128 block read	4.736	0.282	5.96	0.282	5.96	0.282	5.96
128 block write	4.835	0.282	5.83	0.282	5.83	0.282	5.83

Table 5.6: $E_{1\rightarrow 2}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=0** μ**s, QLogic.**

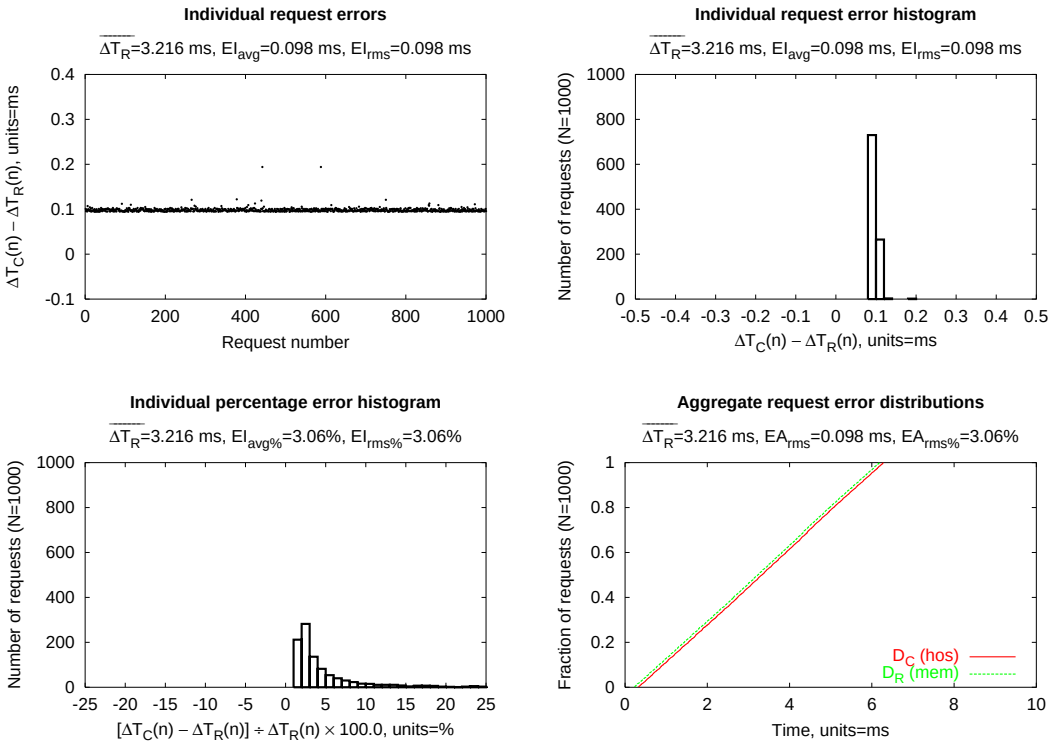


Figure 5.17: $E_{1\rightarrow2}$**:** $\Delta T_{lookahead}$**=0** μ**s,** ΔT_{skew}**=0** μ**s, Adaptec, 4 KB Reads.**

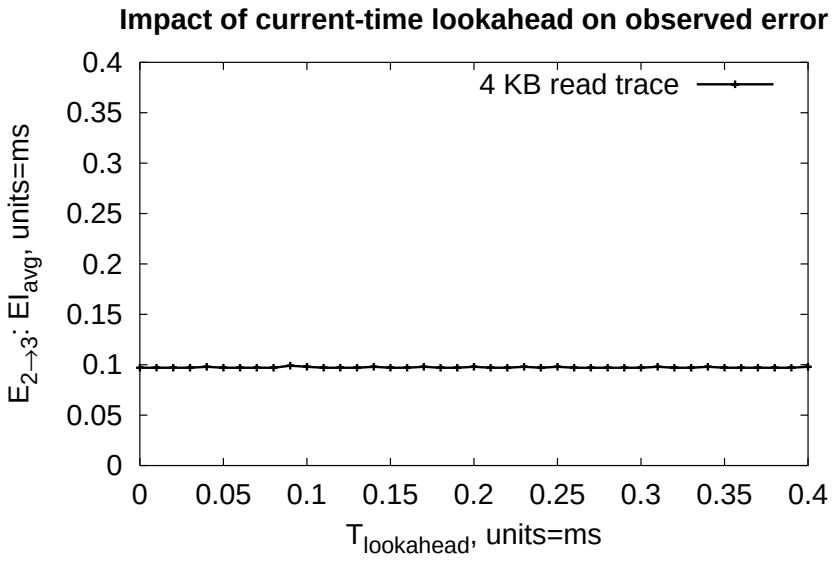


Figure 5.18: The effect of $\Delta T_{lookahead}$ **on** $E_{1\rightarrow2}$**, Adaptec.** *Note that the scale of the X- and Y-axes have changed from the previous graphs.*

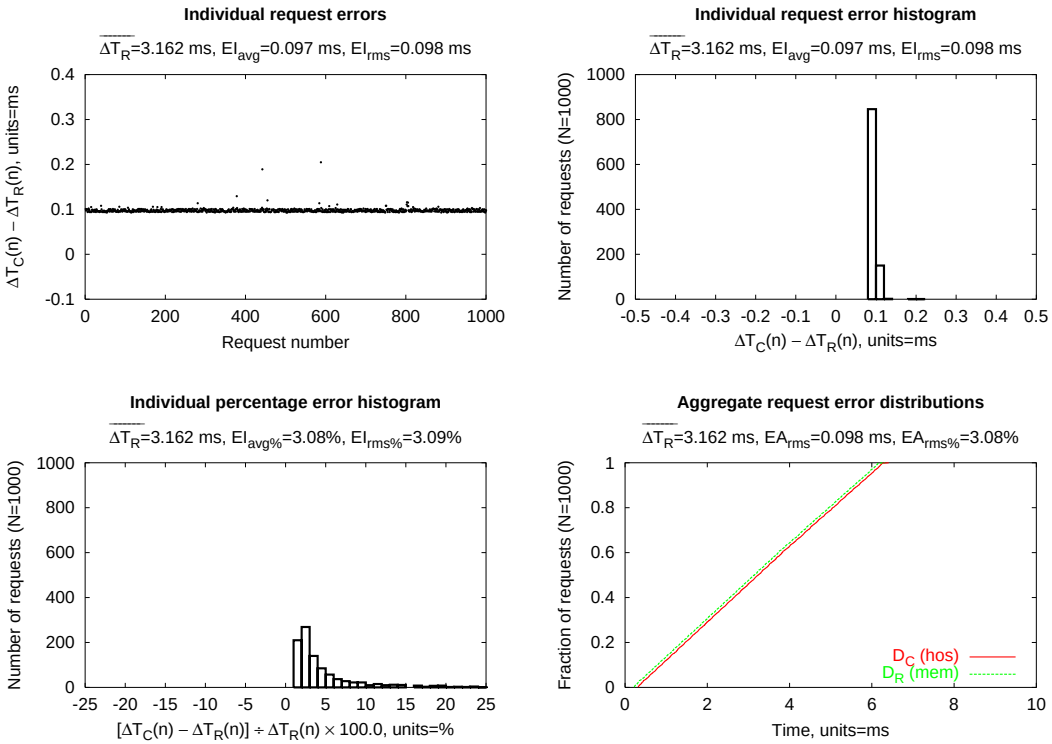


Figure 5.19: $E_{1\rightarrow2}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=0** μ**s, Adaptec, 4 KB Reads.**

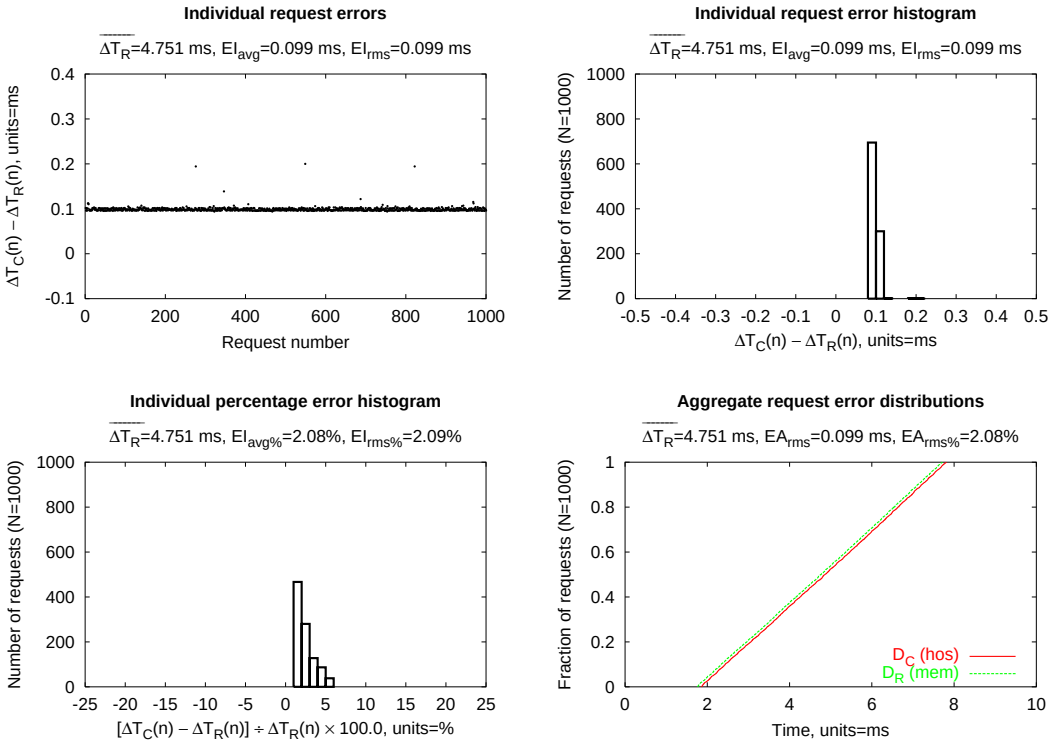


Figure 5.20: $E_{1\rightarrow2}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=0** μ**s, Adaptec, 64 KB Reads.**

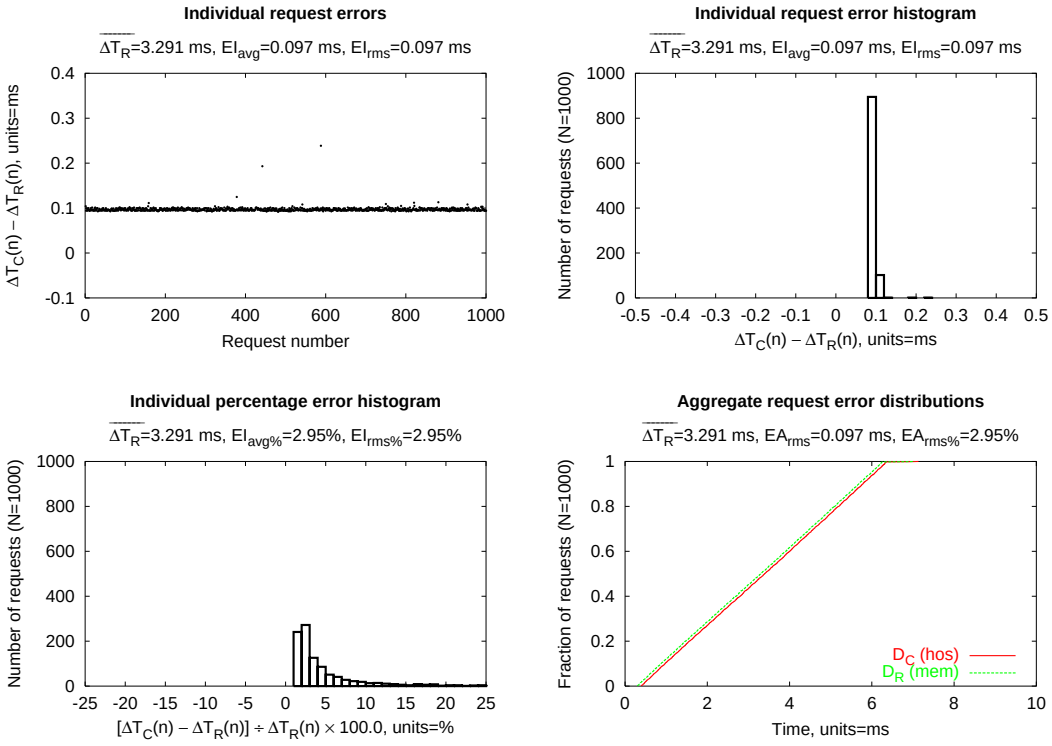


Figure 5.21: $E_{1\rightarrow2}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=0** μ**s, Adaptec, 4 KB Writes.**

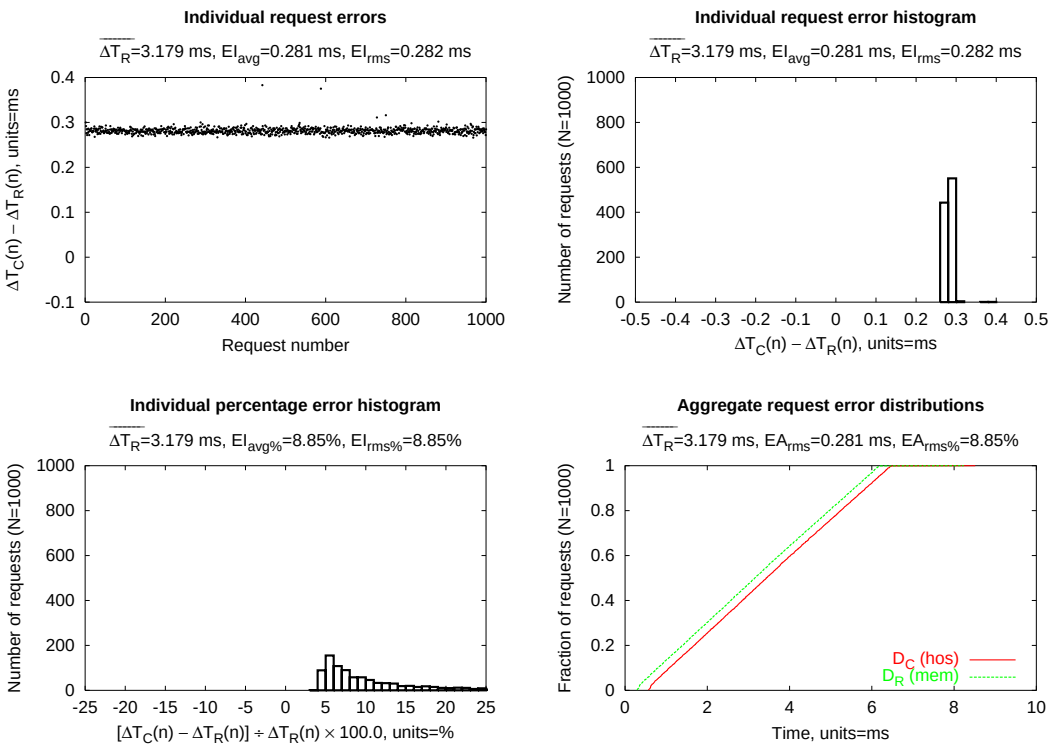


Figure 5.22: $E_{1\rightarrow 2}$: $\Delta T_{lookahead}$=0 μs, ΔT_{skew}=0 μs, QLogic, 4 KB Reads.

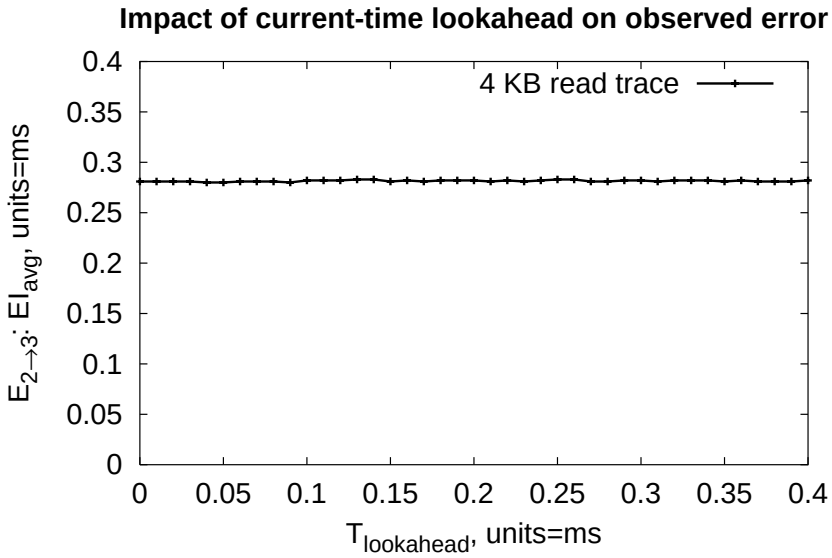


Figure 5.23: The effect of $\Delta T_{lookahead}$ on $E_{1\rightarrow 2}$, QLogic.

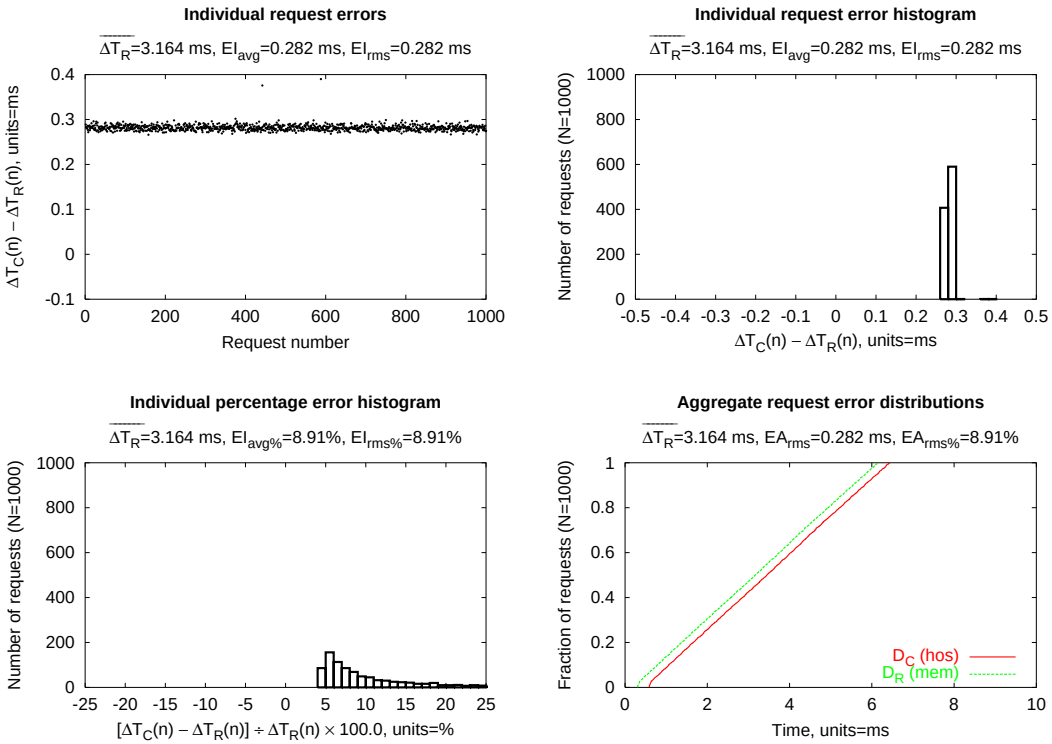


Figure 5.24: $E_{1\rightarrow2}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=0** μ**s, QLogic, 4 KB Reads.**

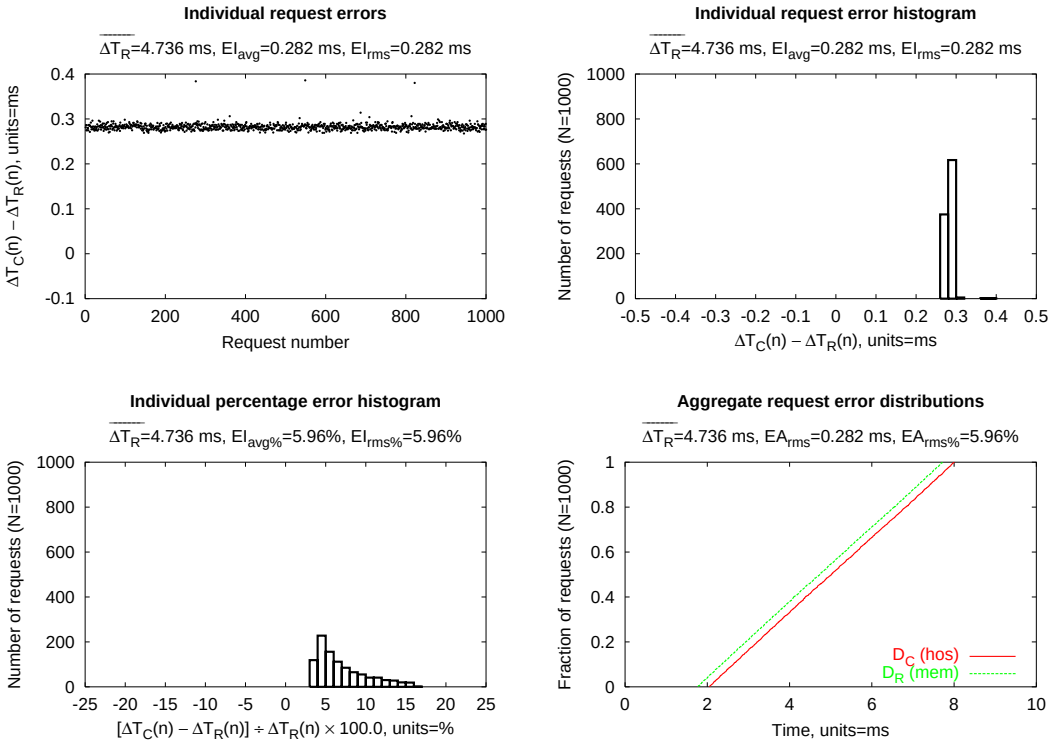


Figure 5.25: $E_{1\rightarrow2}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=0** μ**s, QLogic, 64 KB Reads.**

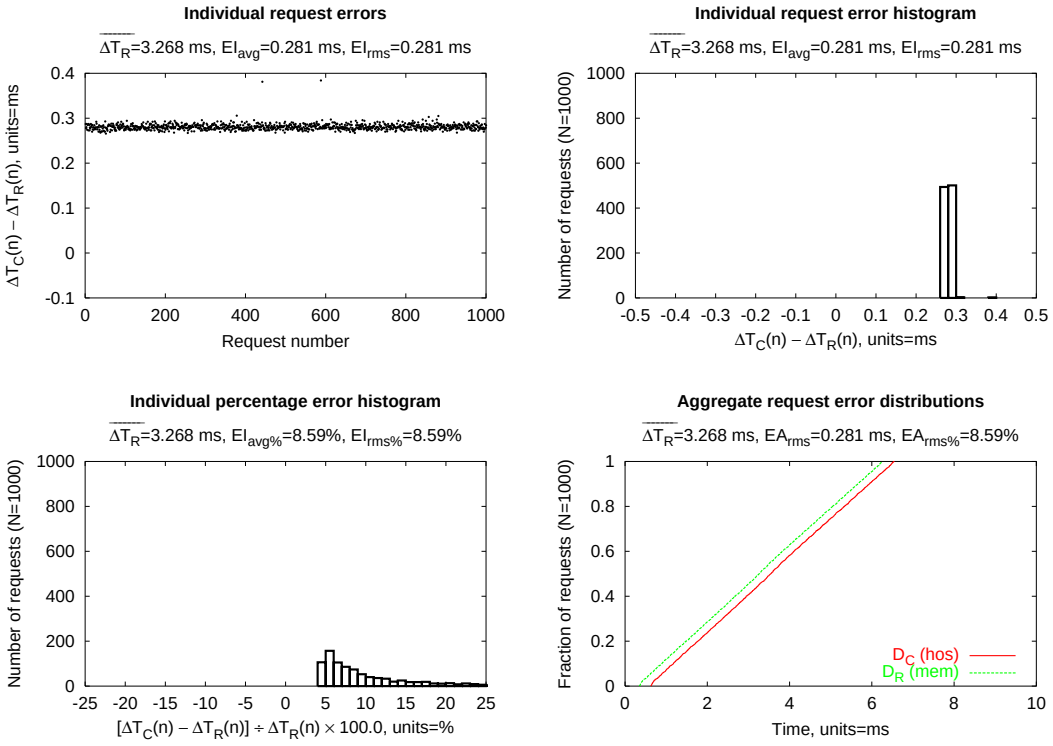


Figure 5.26: $E_{1\rightarrow2}$: $\Delta T_{lookahead}$=30 μs, ΔT_{skew}=0 μs, QLogic, 4 KB Writes.

5.4.3 Mitigation of the quantified response time errors

After determining empirical values for $E_{1\rightarrow 2}$ (Equation 5.13, page 77) and $E_{2\rightarrow 3}$ (Equation 5.12, page 68), we applied Equation 5.1 (page 53) to determine the value $E_{1\rightarrow 3}$. When setting the value of $\Delta T_{lookahead}$=30, the expected per-request error for our implementation is 119 μs when using the Adaptec HBA and 302 μs when using the QLogic HBA.

To verify that the desired results were obtained through this approach, we configured the emulator with an appropriate ΔT_{skew}value (Equation 5.10, page 61)—60 μs when using the Adaptec HBA and 151 μs when using the QLogic HBA—and replayed each of the experimental traces against the emulator. The full results for the experiments in this section, representing all eight traces for both the Adaptec and QLogic bus adapters, are presented in Table 5.7 and Table 5.8. As before, the data in the tables and figures in this section represent a composite of the best values after executing each trace three times. Graphs showing the results of individual experiments are presented as follows:

- For the Adaptec HBA: 4 KB reads (Figure 5.27), 64 KB reads (Figure 5.28), and 4 KB writes (Figure 5.29).
- For the QLogic HBA: 4 KB reads (Figure 5.30), 64 KB reads (Figure 5.31), and 4 KB writes (Figure 5.32).

These data show that the request errors are mitigated to within a maximum EI_{avg}=0.007 ms ($EI_{avg\%}$=0.21%). As is evident in the figures, this approach does not eliminate all sources of inaccuracy in the emulation environment. 5initiation of data transfers is discussed in As seen later, a higher traffic volume with short interrequest delays appears to exacerbate this problem. We expect that continued development on target-mode support in operating system device drivers and bus adapter firmware will further mitigate these problems in future experimentation.

$\Delta T_C \rightarrow \Delta T_R$ (hos→sim)	$\overline{\Delta T_R}$ [ms]	EI_{avg} [ms]	$EI_{avg\%}$ [%]	EI_{rms} [ms]	$EI_{rms\%}$ [%]	EA_{rms} [ms]	$EA_{rms\%}$ [%]
Adaptec							
2 block read	3.137	0.001	0.02	0.072	2.29	0.013	0.41
2 block write	3.229	0.007	0.22	0.111	3.45	0.015	0.45
8 block read	3.221	0.003	0.10	0.120	3.72	0.014	0.43
8 block write	3.301	0.002	0.07	0.081	2.44	0.009	0.27
64 block read	3.945	0.000	0.00	0.087	2.20	0.008	0.21
64 block write	4.047	0.007	0.17	0.095	2.34	0.061	1.51
128 block read	4.777	0.002	0.04	0.087	1.81	0.061	1.28
128 block write	4.860	0.004	0.09	0.048	0.99	0.036	0.74

Table 5.7: $E_{1\rightarrow 3}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=60** μ**s, Adaptec.**

$\Delta T_C \rightarrow \Delta T_R$ (hos→sim)	$\overline{\Delta T_R}$ [ms]	EI_{avg} [ms]	$EI_{avg\%}$ [%]	EI_{rms} [ms]	$EI_{rms\%}$ [%]	EA_{rms} [ms]	$EA_{rms\%}$ [%]
Qlogic							
2 block read	3.239	0.006	0.17	0.048	1.49	0.035	1.09
2 block write	3.336	0.006	0.17	0.029	0.88	0.027	0.81
8 block read	3.294	0.004	0.13	0.040	1.20	0.034	1.04
8 block write	3.428	0.007	0.21	0.024	0.71	0.022	0.64
64 block read	4.023	–0.002	–0.04	0.011	0.26	0.006	0.14
64 block write	4.146	0.005	0.13	0.024	0.58	0.008	0.20
128 block read	4.880	0.003	0.06	0.064	1.32	0.008	0.17
128 block write	4.963	0.010	0.21	0.076	1.52	0.013	0.27

Table 5.8: $E_{1\rightarrow 3}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=151** μ**s, QLogic.**

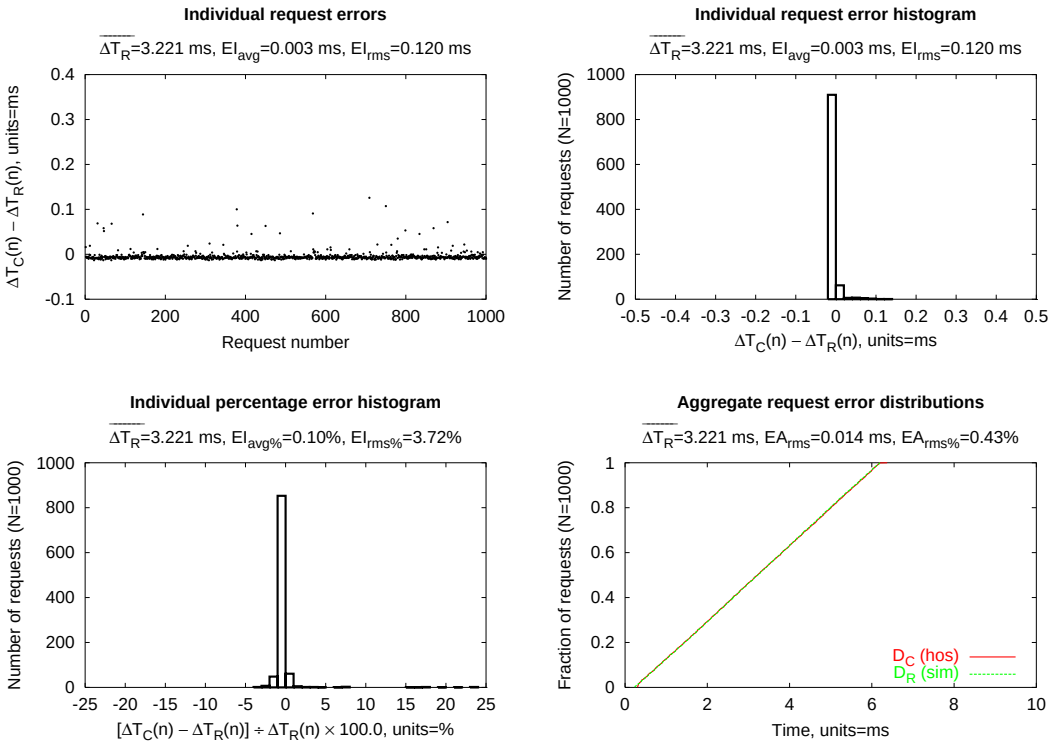


Figure 5.27: $E_{1\rightarrow3}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=60** μ**s, Adaptec, 4 KB Reads.**

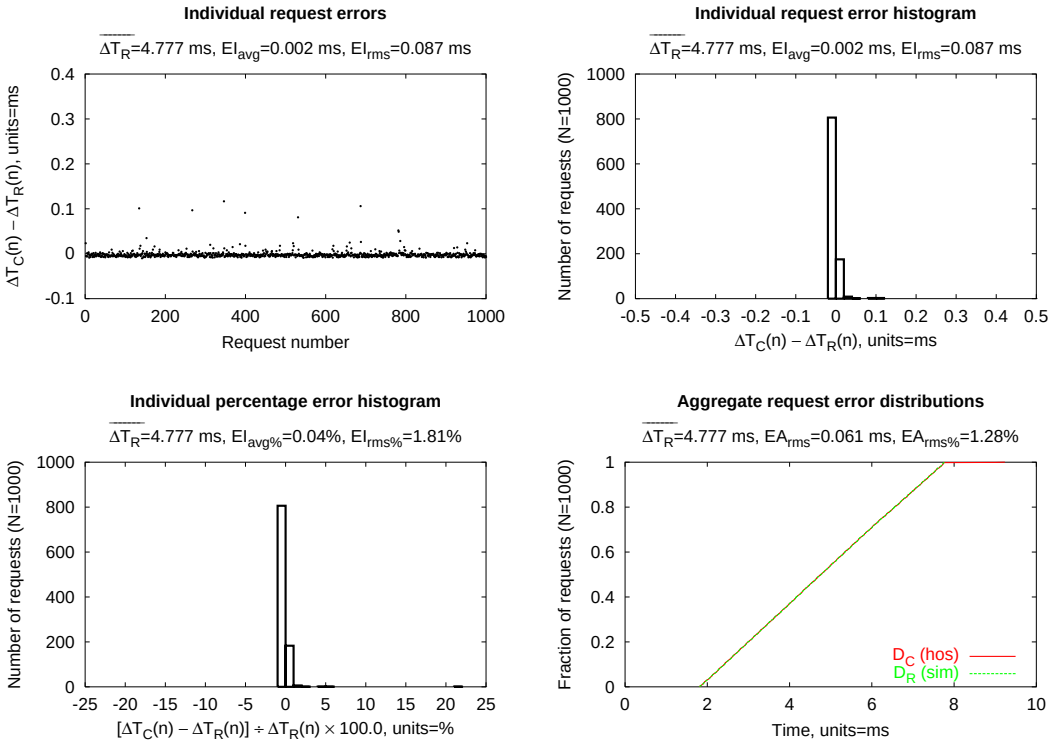


Figure 5.28: $E_{1\rightarrow3}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=60** μ**s, Adaptec, 64 KB Reads.**

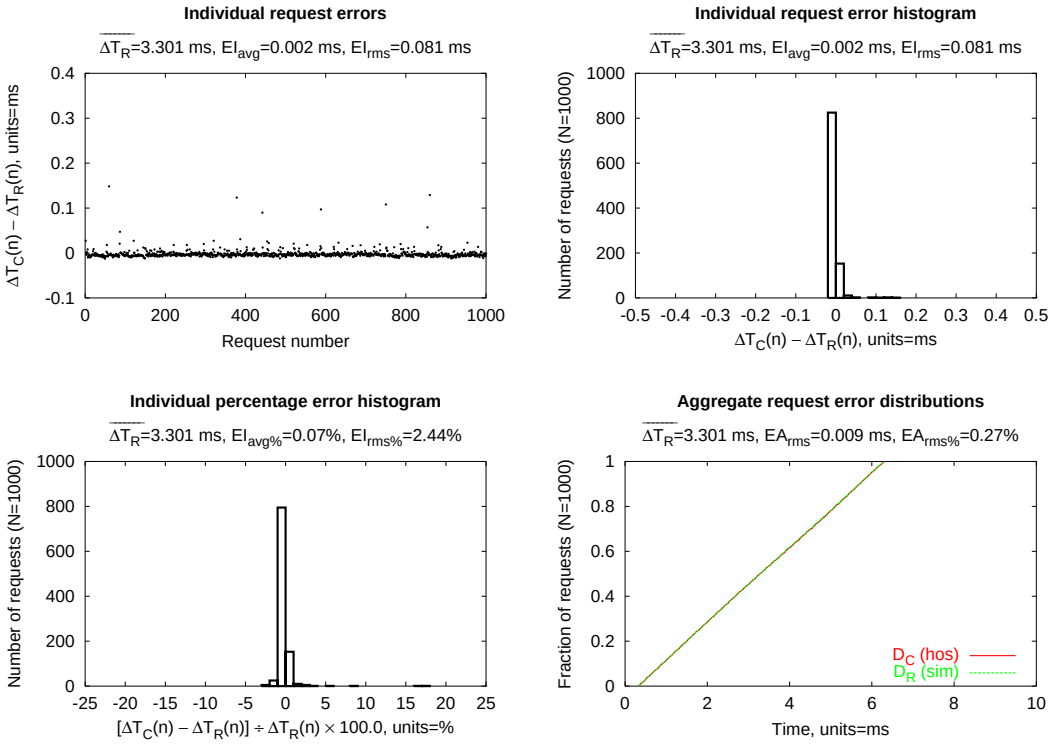


Figure 5.29: $E_{1\rightarrow3}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=60** μ**s, Adaptec, 4 KB Writes.**

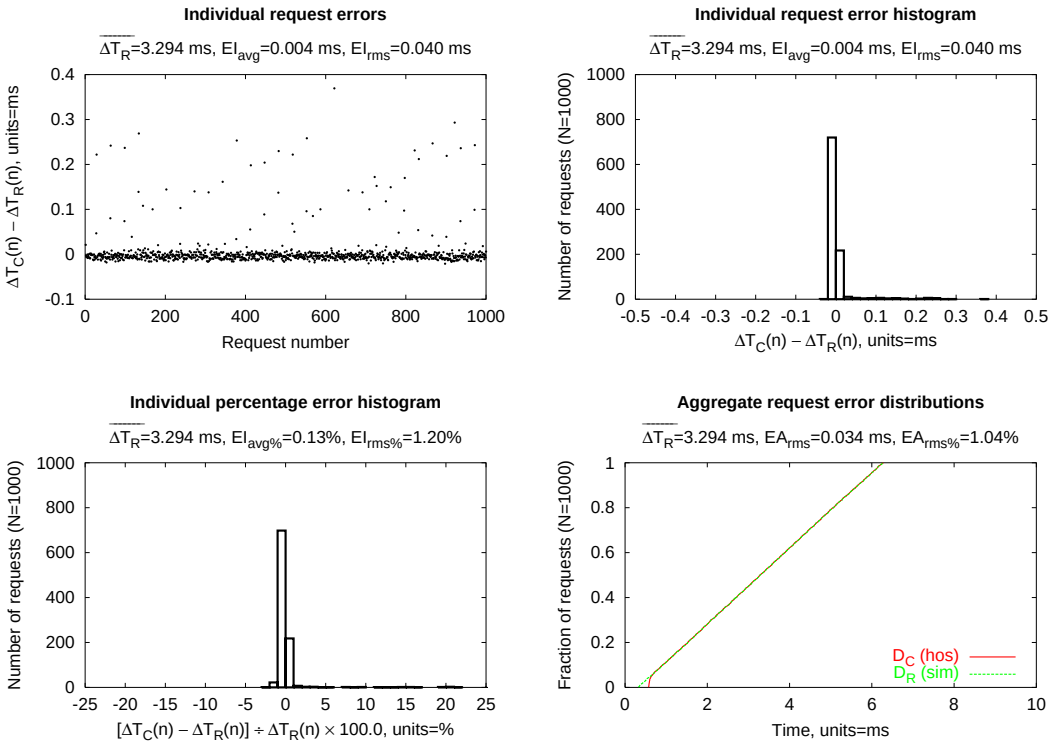


Figure 5.30: $E_{1\rightarrow3}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=151** μ**s, QLogic, 4 KB Reads.**

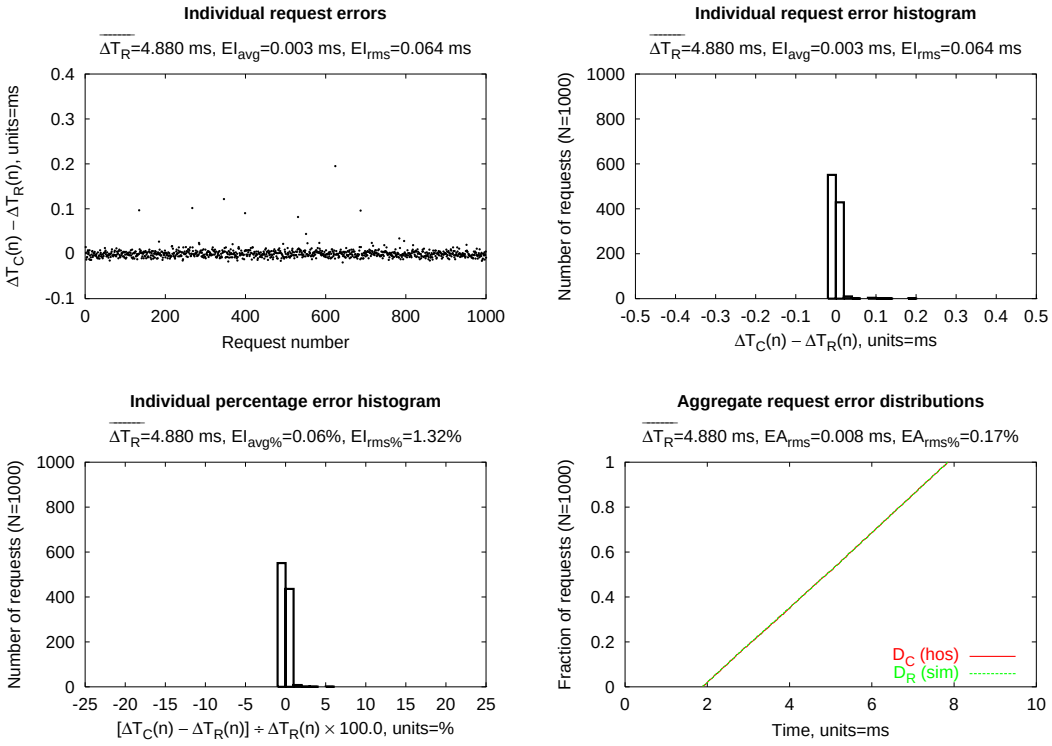


Figure 5.31: $E_{1\rightarrow3}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=151** μ**s, QLogic, 64 KB Reads.**

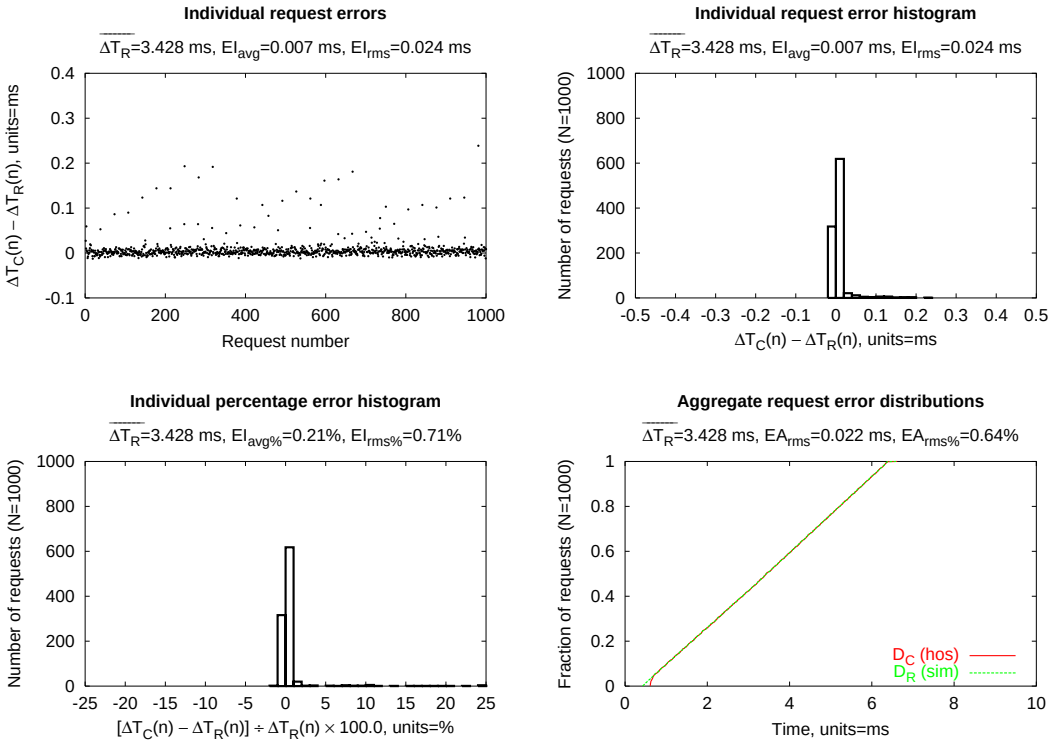


Figure 5.32: $E_{1\rightarrow3}$: $\Delta T_{lookahead}$=30 μs, ΔT_{skew}=151 μs, QLogic, 4 KB Writes.

5.5 Summary of this chapter

The design of the timing manager in a timing-accurate storage emulator involves techniques for equalizing the request service times between the emulated device model and a measurement point elsewhere in the computer system. Analysis of collected per-request service times can be used to calibrate an emulator's operation and mitigate future errors. Very accurate results were obtained experimentally when applying error compensation techniques to our implementation of a timing-accurate storage emulator.

The following chapter demonstrates a strength of timing-accurate storage emulation in evaluating system architectures containing hypothetical or otherwise unavailable storage components.

CHAPTER 6
EXPERIMENTS WITH HYPOTHETICAL STORAGE DEVICES

This chapter reports on the experimental setup (Section 6.1) and results of system-level experimentation and experimental validation (Section 6.2, page 96) using timing-accurate storage emulation of device models for both existing and hypothetical storage devices. The chapter additionally validates the feasibility of timing-accurate storage emulation by comparing the performance of an emulator with that of a real storage device (Section 6.3, page 110).

6.1 Experimental setup

The host and emulator systems were configured as described in Chapter 4, specifically using the Adaptec 29160 parallel SCSI adapters to connect the systems. Each application-level experiment was executed ten times; the measurements are reported both across all runs (the mean and standard deviation of each metric) and specifically for the emulated run with the EI_{avg} metric closest to zero. In an effort to exercise the worst-case storage performance for the experiments, the experimental devices (both real and emulated) are mounted synchronously by the host computer and caching was disabled in the device and all device models.

6.1.1 Device models used for experimentation

Each of the application-level experiments are run against one real disk and four emulated device configurations, as described in this section.

1. *Real Cheetah.* Each experiment is first run against a local disk on the host system, as a reference for comparison of the run times under emulation. A Seagate Cheetah disk was used for this purpose. The characteristics of this disk are described in Table 4.1 on page 44; of importance to the models below is that the disk platters rotate at 10,000 revolutions per minute (RPM). The disk was connected to the host system using approach (1) illustrated in Figure 4.1 on page 43.

2. *Emulated Cheetah.* The experiments are repeated using timing-accurate storage emulation of a device model of the Real Cheetah disk. This model was previously validated to an EA_{rms} value of 0.15 ms, as shown in the aggregate distribution in Figure 6.1.

3. *Emulated 50K RPM.* To explore the impact of modifying a device in a way that is currently physically impossible, we scaled the physical characteristics of the Emulated Cheetah model to create a 50,000 RPM disk. Specifically, we reduced the head switch time and increased the rotation speed each by a factor of five.

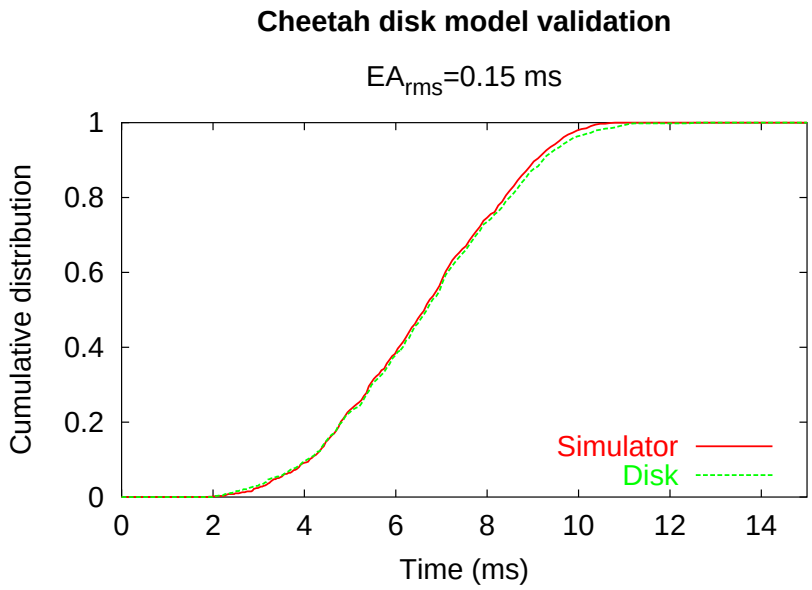


Figure 6.1: Validation of the disk model for the Seagate Cheetah ST336706LC. *These data were generated by John Bucy using a synthetically-generated random workload consisting 66% read requests, an exponential request size distribution with mean 8 KB, and request locations uniformly distributed across the disk capacity. Note that these data were not obtained using the specific disk used in our experimentation, but rather another disk of the same product line.*

4. *Emulated MEMS*. To explore the impact of storage devices that do not yet exist, we built a simulation model of a MEMS-based storage device. As described in Appendix A, MEMS-based storage devices are a novel technology under development at several research laboratories and companies around the world. These devices are expected to perform faster than disk drives but have much smaller capacities than disks.

5. *Emulated Immediate*. To verify that the Emulated 50K RPM and Emulated MEMS experiments are not operating at the boundary of possible performance of our emulator, we configured the emulator to respond immediately to requests as soon as the request data are available in the data manager. Note that due to the architecture of our implementation—specifically, the data manager runs as a separate process from the communications and timing managers—the experiments do not represent the absolute best performance that could be achieved with our hardware configuration.

6.1.2 Description of experimental workloads

Three application-level workloads were used for our experiments, as described in this section. The first two, PostMark and SSH-build, are based on previously-published file system benchmarks. The third, Linux kernel build, was created for this dissertation.

6.1.2.1 PostMark benchmark

The *PostMark* application-level benchmark was introduced by Katcher in 1997 as a method to recreate file system operations that are representative of those generated by electronic mail, newsgroup, and web-based services [96]. The benchmark 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; in our experiments, we use the default PostMark configuration with no user-specified weights. We executed the benchmark using the following procedure:

- *Preparation phase.*
  Make a fresh ext2 file system on the emulated device (`/dev/sdd1`).

- *Execution phase.*

```
mount -osync /dev/sdd1 /mnt/memulator
cp /exp/postmark /exp/postmark.config /mnt/memulator
cd /mnt/memulator
/usr/bin/time ./postmark postmark.config
cd /exp
umount /mnt/memulator
```

6.1.2.2 Secure shell (SSH) build benchmark

The *SSH-build* application-level benchmark was introduced by Seltzer et al. in 2000 as a method of stressing a storage subsystem using a variety of file system operations [151]. Values are reported representing the three benchmark phases: unarchiving the source code[6.1] of the SSH "secure shell" utility, running a utility to examine the host system and prepare the source code for site-specific compilation, and compiling the source code into the SSH binary executable and affiliated library files. We executed the benchmark using the following procedure:

- *Warmup phase.*
  Make a fresh ext2 file system on the emulated device (`/dev/sdd1`). Execute the unpack, configure, and build phases in the temporary (`/tmp`) directory on the host system.

[6.1] Seltzer et al. used a `gzip`-compressed `tar` archive of the source code for SSH version 1.2.26; for historical reasons, we use an uncompressed `tar` archive of SSH version 1.2.27. We also mount the emulated device synchronously, and unmount the emulated device between benchmark phases.

- *Unpack phase.*

```
mount -osync /dev/sdd1 /mnt/memulator
cd /mnt/memulator
/usr/bin/time tar xf /exp/ssh-1.2.27.tar > /dev/null 2>&1
cd /exp
umount /mnt/memulator
```

- *Configure phase.*

```
mount -osync /dev/sdd1 /mnt/memulator
cd /mnt/memulator/ssh-1.2.27
/usr/bin/time ./configure --without-x > /dev/null 2>&1
cd /exp
umount /mnt/memulator
```

- *Build phase.*

```
mount -osync /dev/sdd1 /mnt/memulator
cd /mnt/memulator/ssh-1.2.27
/usr/bin/time make > /dev/null 2>&1
cd /exp
umount /mnt/memulator
```

6.1.2.3 Linux kernel build

We use the compilation of the Linux 2.4.27 operating system kernel as an additional experiment representing a much more commonly executed application-level workload[6.2] than the SSH-build benchmark. This *Linux-build* application-level benchmark offers an interesting alternation between begin storage-bound and processor-bound during the experiment. The class of benchmarks represented by Linux-build is intended to compare the impact of storage decisions using identical external system configurations—i.e., comparing a disk and a MEMS-based storage device when each are connected to a 2 GHz processor-based system—and not necessarily to provide comparable results between different external system configurations.

Due to the interactive nature of the Linux kernel build configuration process (we are not aware of a method for compiling a default kernel without first running one of the manual-entry configuration scripts), we omit the unpack and configure phases as are used in SSH-build. Instead, we created a disk image representing the default post-configuration state. This was accomplished by running `make menuconfig` and exiting immediately, saving the configuration. This disk image was loaded onto the emulated device before the device was mounted, and the benchmark only reports the timing for the make phase. We executed the benchmark using the following procedure:

[6.2] We note tongue-in-cheek that operating system kernel recompilation is perhaps one of the most commonly executed workloads for experimenters who may be interested in timing-accurate storage emulation.

- *Preparation phase.* Load the disk image onto the emulated device (`/dev/sdd1`) as described above.
- *Build phase.*

```
mount -osync /dev/sdd1 /mnt/memulator
cd /mnt/memulator/linux-2.4.27
/usr/bin/time make > /dev/null 2>&1
cd /exp
umount /mnt/memulator
```

6.2 Results of experimentation

This section reports on the results of our experimentation with the real disk and four emulator configurations introduced in Section 6.1.1. An initial summary containing a comparison of all average application-level run times is presented in Table 6.1. The remainder of the section is then organized as follows:

Tables are provided for each application-level workload showing the average quantitative metrics across all experimental runs, as well as the metrics for the three best individual runs with the lowest individual error (one for each emulated device model). Detailed data for these three best runs are shown in the figures immediately following the table of results. Each figure contains four fixed-axis subgraphs, identical to the subgraphs in the figures of Section 5.4 (page 66). Each trace was experimentally executed three times, and the subgraphs and quantitative metrics presented in each figure represent the best or lowest values obtained across the three experiments. Each of the graphs displayed here have fixed X- and Y-axes to aid in comparison between graphs. This caused an abridgment of some of the extreme outliers on the graphs, so we provide full graphs of individual request errors for each experiment in Appendix D.

Real Cheetah [s]		Emulated Cheetah [s]		Emulated 50K RPM [s]		Emulated MEMS [s]		Emulated Immediate [s]	
PostMark									
46.21	*(0.03)*	45.99	*(0.12)*	15.85	*(0.07)*	11.42	*(0.03)*	3.38	*(0.90)*
		99.5%		34.3%		24.7%		7.3%	
SSH-build									
217.34	*(0.20)*	218.27	*(0.43)*	107.97	*(0.28)*	82.65	*(0.34)*	51.55	*(0.21)*
		100.4%		49.7%		38.0%		23.7%	
(unpack phase)									
25.24	*(0.04)*	25.80	*(0.06)*	9.30	*(0.01)*	6.04	*(0.03)*	1.77	*(0.01)*
		102.2%		36.8%		23.9%		7.0%	
(configure phase)									
54.38	*(0.12)*	54.79	*(0.21)*	26.44	*(0.17)*	21.67	*(0.11)*	12.50	*(0.04)*
		100.8%		48.6%		39.9%		23.0%	
(make phase)									
137.72	*(0.19)*	137.68	*(0.42)*	72.23	*(0.17)*	54.93	*(0.33)*	37.28	*(0.21)*
		100.0%		52.4%		39.9%		27.1%	
Linux-build									
535.32	*(0.90)*	534.95	*(0.55)*	400.61	*(0.81)*	359.36	*(0.25)*	326.09	*(0.26)*
		99.9%		74.8%		67.1%		60.9%	

Table 6.1: Summary and comparison of mean run times across all experiments. *All non-percentage values are reported in seconds. Each experiment was executed ten times against each real or emulated device, which was mounted synchronously by the host system. The first number represents the mean run time of the experiment. The values in parentheses represent the standard deviation of the run time. The values reported as percentages compare the mean run time of each emulated experiment against the mean run time of the Real Cheetah.*

6.2.1 PostMark benchmark

The experimental methodology for the PostMark benchmark is introduced in Section 6.1.2.1. During execution, we observed this benchmark write approximately 41,800 KB of data and read 93 KB of data. The results for the PostMark-based experimentation presented in this section include:

- The full results across all configurations (Table 6.2).
- Data for the best individual run of the Emulated Cheetah model (Figure 6.2).
- Data for the best individual run of the Emulated 50K RPM model (Figure 6.3).
- Data for the best individual run of the Emulated MEMS model (Figure 6.4).

As discussed in Section 5.4.3, we attribute much of the error to the delays in the target-mode adapter firmware and device driver software in quickly scheduling and completing transfers of data across the storage interconnect.

We additionally speculate that these errors may be caused by caching in the host system device driver. Support for target-mode operation of the Adaptec HBA is limited in that it can only accept one request at a time from the host, as discussed in Section 7.5.1. If the application workload triggers multiple storage requests simultaneously in the host (e.g., data prefetching by the host operating system), some of them will be be queued extra time at measurement point MP_1 before being received at MP_2. This may be a primary source of perceived "error" (but not necessarily real error) in our results. Potential workarounds for this include locating MP_1 further down in the device driver[6.3] or eliminate MP_1 altogether, as suggested for future exploration in Section 8.4.3 (page 140).

[6.3] There is a trade-off between this problem and the flexibility and ease of inserting a measurement point. It is architecturally straightforward to insert a measurement point at the Linux SCSI mid-to-low-layer boundary, as we do for MP_1, but much less straightforward to do so in the low-level device driver.

		Real Cheetah	Emulated Cheetah	Emulated 50K RPM	Emulated MEMS	Emulated Immediate
		Application run times (averaged across all runs)				
Run time	s	46.21 *(0.03)*	45.99 *(0.12)*	15.85 *(0.07)*	11.42 *(0.03)*	3.38 *(0.90)*
Minimum run time	s	46.17	45.79	15.74	11.38	3.08
Maximum run time	s	46.26	46.15	15.96	11.47	5.93
		Request comparison metrics (averaged across all runs)				
Total requests	#	10421	10422	10421	10421	10421
$\overline{\Delta T_C}$ (in host)	ms	4.477 *(0.003)*	4.456 *(0.010)*	1.530 *(0.008)*	1.096 *(0.004)*	0.313 *(0.089)*
$\overline{\Delta T_R}$ (in simulator)	ms		4.381 *(0.002)*	1.471 *(0.002)*	1.080 *(0.001)*	
EI_{avg}	ms		0.075 *(0.010)*	0.059 *(0.009)*	0.015 *(0.004)*	
$EI_{avg\%}$	%		1.7 *(0.2)*	4.0 *(0.6)*	1.4 *(0.4)*	
EI_{rms}	ms		0.991 *(0.303)*	0.888 *(0.454)*	0.344 *(0.220)*	
$EI_{rms\%}$	%		22.6 *(6.9)*	60.4 *(30.9)*	31.9 *(20.4)*	
EA_{rms}	ms		0.860 *(0.330)*	0.838 *(0.463)*	0.316 *(0.227)*	
$EA_{rms\%}$	%		19.6 *(7.5)*	57.0 *(31.5)*	29.3 *(21.0)*	
		Request comparison metrics (for the run with the best EI_{avg})				
Run time	s		45.79	15.79	11.41	
Total requests	#		10422	10422	10422	
$\overline{\Delta T_C}$ (in host)	ms		4.442	1.522	1.091	
$\overline{\Delta T_R}$ (in simulator)	ms		4.382	1.473	1.081	
EI_{avg}	ms		0.061	0.050	0.011	
$EI_{avg\%}$	%		1.4	3.4	1.0	
EI_{rms}	ms		0.560	0.621	0.132	
$EI_{rms\%}$	%		12.8	42.2	12.2	
EA_{rms}	ms		0.380	0.577	0.087	
$EA_{rms\%}$	%		8.7	39.2	8.1	

Table 6.2: Detailed results for all PostMark experiments.

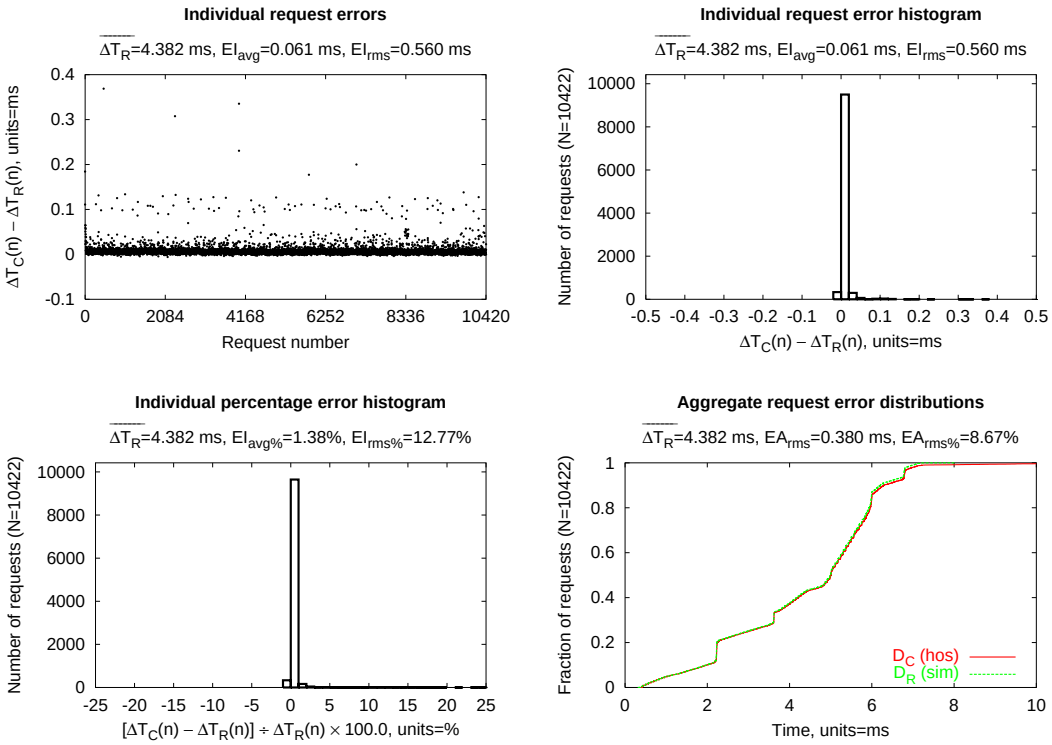


Figure 6.2: PostMark: Individual results for the Emulated Cheetah model.

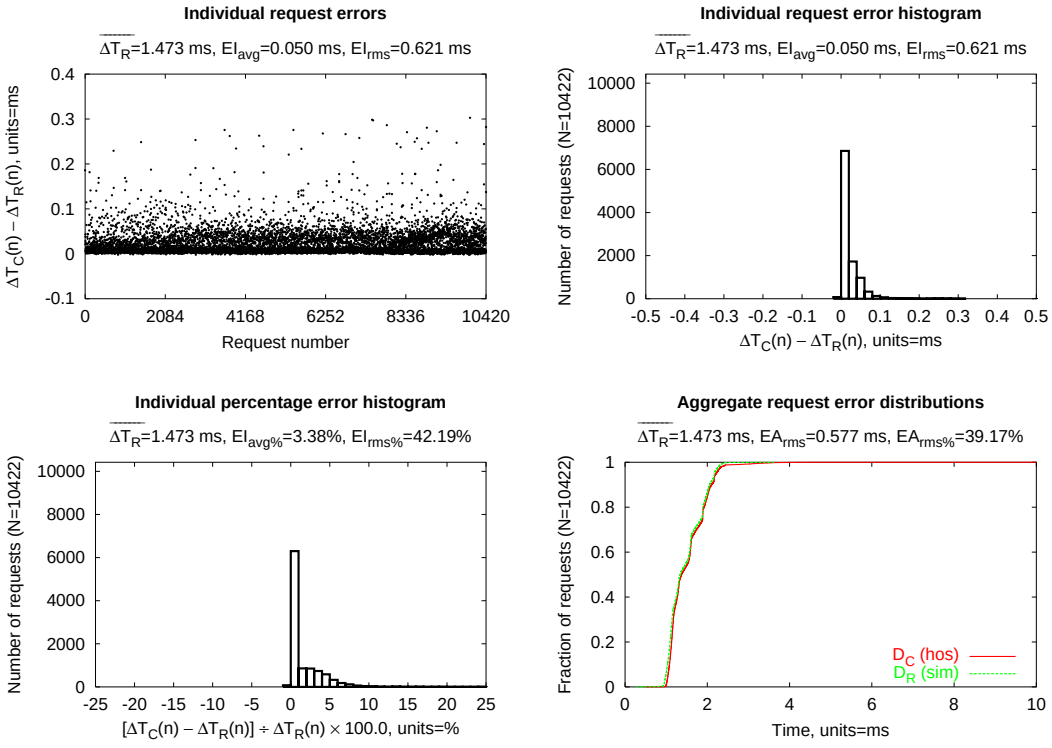


Figure 6.3: PostMark: Individual results for the Emulated 50K RPM model.

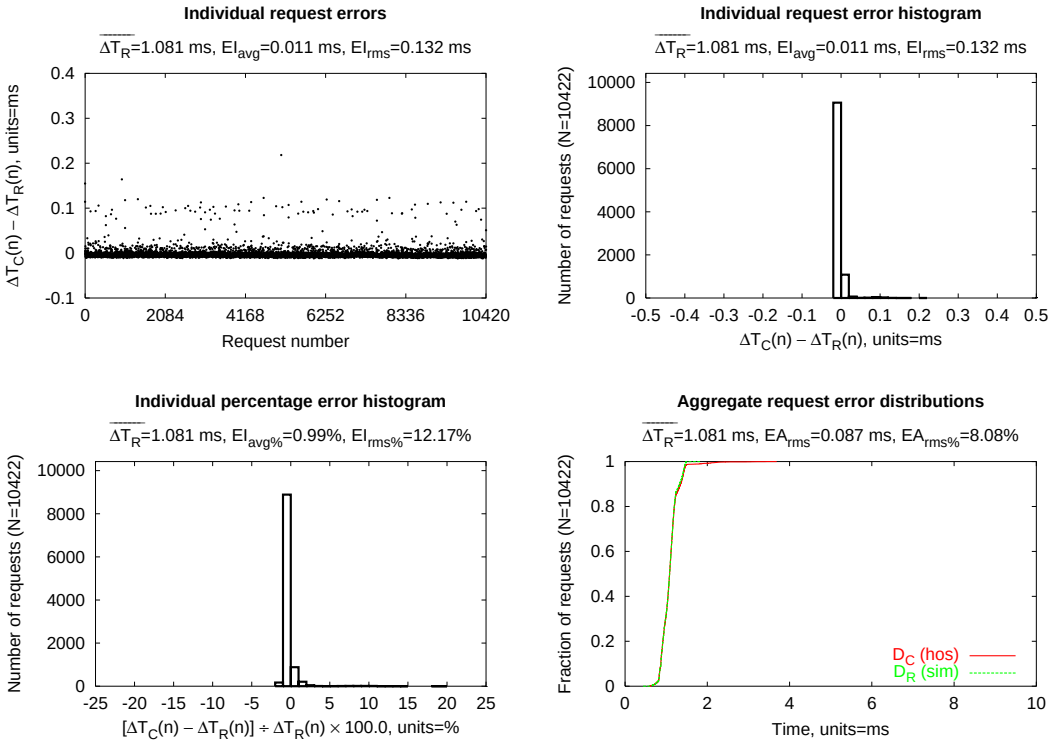


Figure 6.4: PostMark: Individual results for the Emulated MEMS model.

6.2.2 Secure shell (SSH) build benchmark

The experimental methodology for the SSH-build benchmark is introduced in Section 6.1.2.2. During execution, we observed this benchmark write approximately 161,500 KB of data and read 4,255 KB of data. The results for the SSH-build-based experimentation presented in this section include:

- The full results across all configurations (Table 6.3).
- Data for the best individual run of the Emulated Cheetah model (Figure 6.5).
- Data for the best individual run of the Emulated 50K RPM model (Figure 6.6).
- Data for the best individual run of the Emulated MEMS model (Figure 6.7).

Due to the warmup phase of our experimental methodology—the `tar` archive of the software is cached by the host operating system—the unpack phase of SSH-build is almost entirely I/O-bound on our emulator, resulting in large reductions in application run-time for the faster device models; these reductions are comparable in time to the results from the PostMark benchmark.

		Real Cheetah	Emulated Cheetah	Emulated 50K RPM	Emulated MEMS	Emulated Immediate
		Application run times (averaged across all runs)				
Run time	s	217.34 *(0.20)*	218.27 *(0.43)*	107.97 *(0.28)*	82.65 *(0.34)*	51.55 *(0.21)*
Minimum run time	s	217.05	217.42	107.51	82.35	51.34
Maximum run time	s	217.59	218.87	108.58	83.48	52.00
		Request comparison metrics (averaged across all runs)				
Total requests	#	39488	39496	39494	39492	39487
$\overline{\Delta T_C}$ (in host)	ms	4.569 *(0.003)*	4.566 *(0.008)*	1.745 *(0.004)*	1.089 *(0.002)*	0.288 *(0.001)*
$\overline{\Delta T_R}$ (in simulator)	ms		4.489 *(0.010)*	1.687 *(0.001)*	1.074 *(0.001)*	
EI_{avg}	ms		0.077 *(0.004)*	0.058 *(0.004)*	0.014 *(0.002)*	
$EI_{avg\%}$	%		1.7 *(0.1)*	3.4 *(0.2)*	1.3 *(0.2)*	
EI_{rms}	ms		0.959 *(0.121)*	0.591 *(0.095)*	0.271 *(0.133)*	
$EI_{rms\%}$	%		21.4 *(2.7)*	35.0 *(5.6)*	25.2 *(12.4)*	
EA_{rms}	ms		0.785 *(0.134)*	0.505 *(0.102)*	0.240 *(0.141)*	
$EA_{rms\%}$	%		17.5 *(3.0)*	29.9 *(6.1)*	22.3 *(13.2)*	
		Request comparison metrics (for the run with the best EI_{avg})				
Run time	s		217.86	107.51	82.35	
Total requests	#		39479	39493	39492	
$\overline{\Delta T_C}$ (in host)	ms		4.565	1.738	1.086	
$\overline{\Delta T_R}$ (in simulator)	ms		4.493	1.686	1.074	
EI_{avg}	ms		0.072	0.051	0.012	
$EI_{avg\%}$	%		1.6	3.0	1.1	
EI_{rms}	ms		0.913	0.420	0.171	
$EI_{rms\%}$	%		20.3	24.9	15.9	
EA_{rms}	ms		0.732	0.315	0.132	
$EA_{rms\%}$	%		16.3	18.7	12.3	

Table 6.3: Detailed results for all SSH-build experiments.

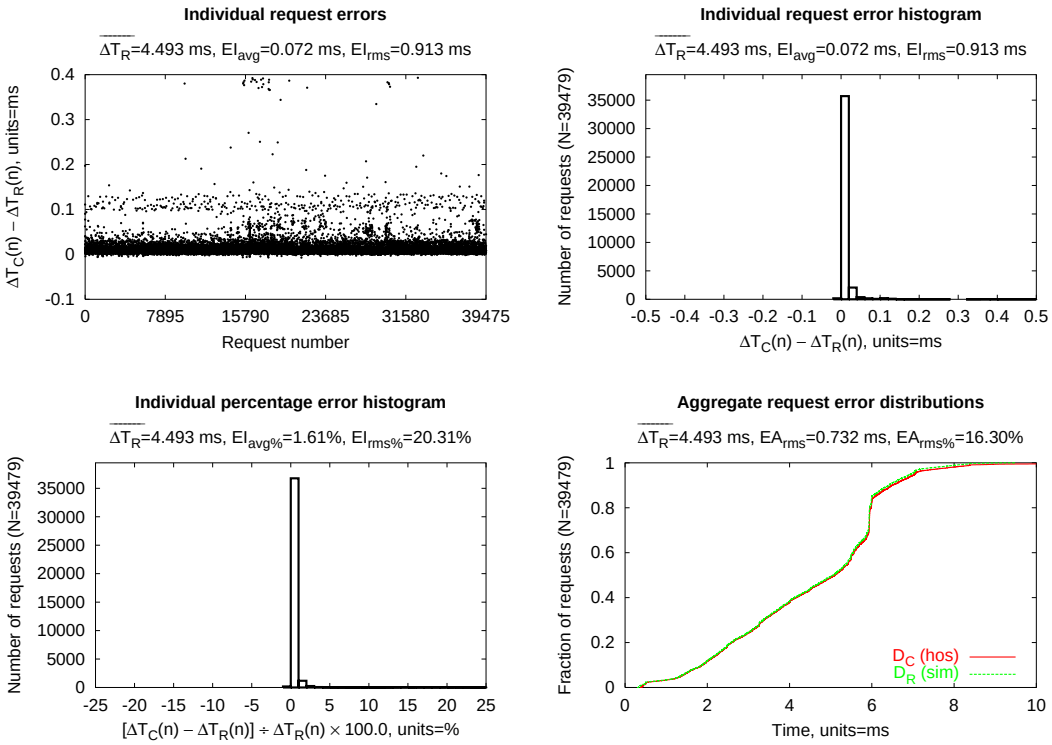


Figure 6.5: SSH-build: Individual results for the Emulated Cheetah model.

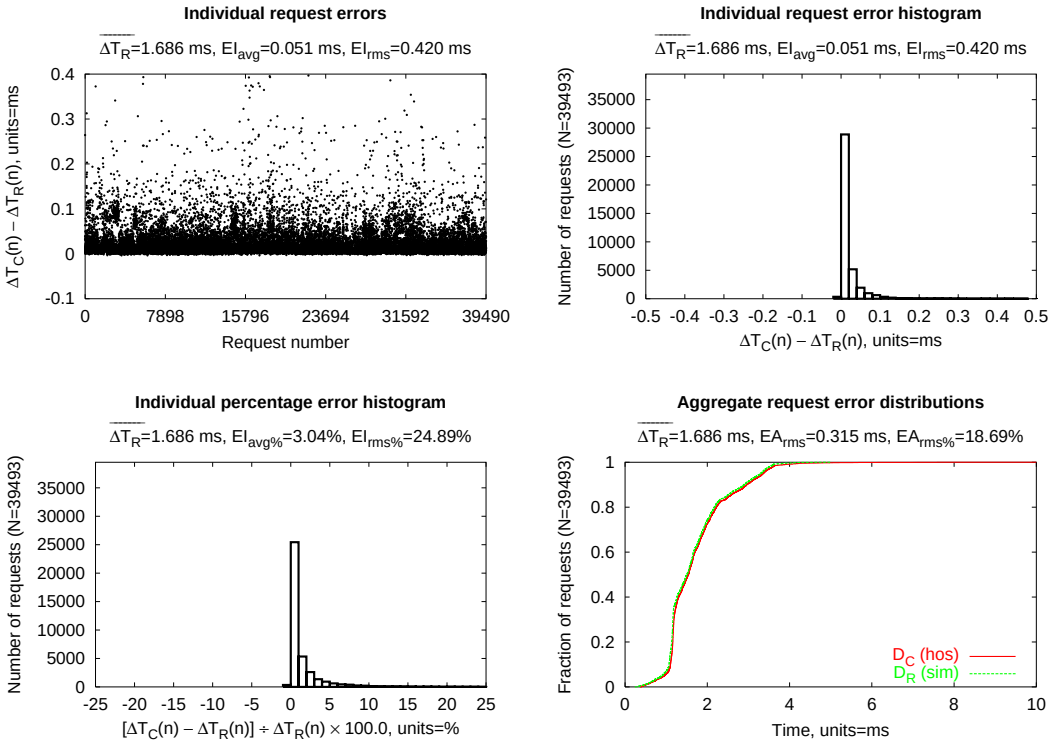


Figure 6.6: SSH-build: Individual results for the Emulated 50K RPM model.

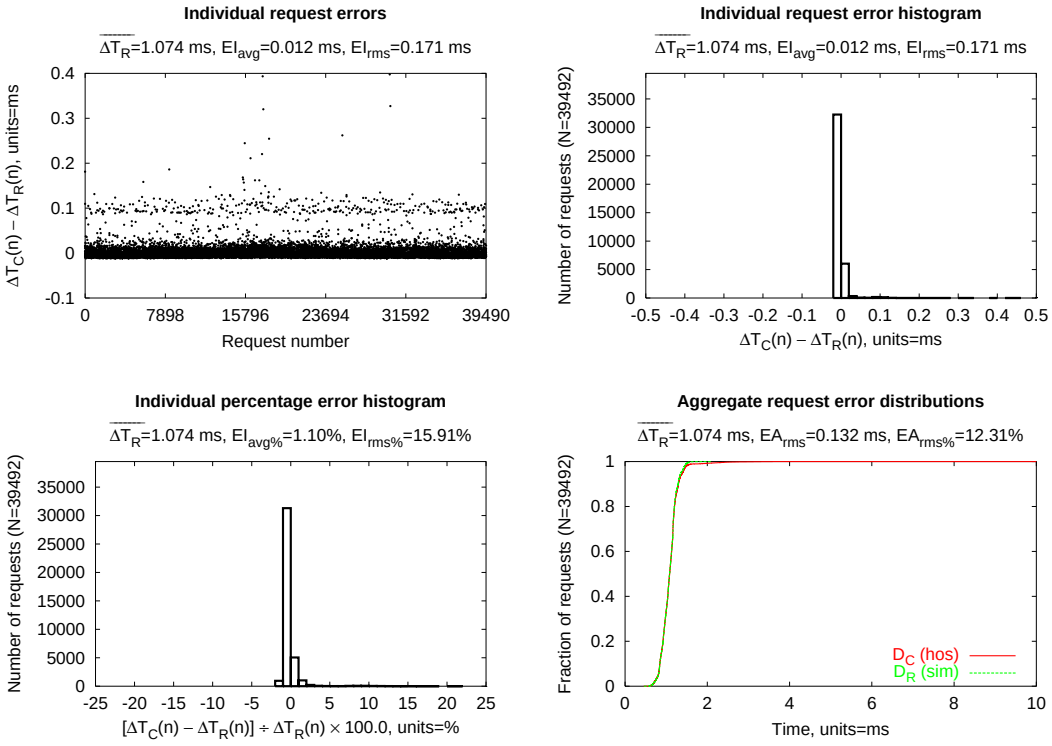


Figure 6.7: SSH-build: Individual results for the Emulated MEMS model.

6.2.3 Linux kernel build

The experimental methodology for the Linux-build experiment is introduced in Section 6.1.2.3. During execution, we observed this benchmark write approximately 166,000 KB of data and read 119,813 KB of data. The results for the Linux-build-based experimentation presented in this section include:

- The full results across all configurations (Table 6.4).
- Data for the best individual run of the Emulated Cheetah model (Figure 6.8).
- Data for the best individual run of the Emulated 50K RPM model (Figure 6.9).
- Data for the best individual run of the Emulated MEMS model (Figure 6.10).

These results show an interesting reduction in the application-level benefit of faster storage device models, as compared with the PostMark and SSH-build benchmarks. This indicates that the Linux kernel compilation spends a proportionately smaller fraction of time I/O-bound than do the benchmarks.

		Real Cheetah	Emulated Cheetah	Emulated 50K RPM	Emulated MEMS	Emulated Immediate
		Application run times (averaged across all runs)				
Run time	s	535.32 *(0.90)*	534.95 *(0.55)*	400.61 *(0.81)*	359.36 *(0.25)*	326.09 *(0.26)*
Minimum run time	s	533.88	534.23	399.78	359.06	325.50
Maximum run time	s	536.92	535.90	402.58	359.71	326.47
		Request comparison metrics (averaged across all runs)				
Total requests	#	46427	46256	46240	46282	46202
$\overline{\Delta T_C}$ (in host)	ms	5.427 *(0.126)*	5.088 *(0.017)*	2.083 *(0.011)*	1.113 *(0.008)*	0.337 *(0.004)*
$\overline{\Delta T_R}$ (in simulator)	ms		4.925 *(0.003)*	1.981 *(0.001)*	1.069 *(0.001)*	
EI_{avg}	ms		0.162 *(0.018)*	0.102 *(0.011)*	0.043 *(0.008)*	
$EI_{avg\%}$	%		3.3 *(0.4)*	5.1 *(0.6)*	4.1 *(0.8)*	
EI_{rms}	ms		1.235 *(0.219)*	1.078 *(0.329)*	0.833 *(0.413)*	
$EI_{rms\%}$	%		25.1 *(4.4)*	54.4 *(16.6)*	77.9 *(38.7)*	
EA_{rms}	ms		0.824 *(0.182)*	0.933 *(0.337)*	0.756 *(0.406)*	
$EA_{rms\%}$	%		16.7 *(3.7)*	47.1 *(17.0)*	70.7 *(38.0)*	
		Request comparison metrics (for the run with the best EI_{avg})				
Run time	s		534.23	399.78	359.24	
Total requests	#		46148	46014	46076	
$\overline{\Delta T_C}$ (in host)	ms		5.064	2.063	1.099	
$\overline{\Delta T_R}$ (in simulator)	ms		4.921	1.982	1.069	
EI_{avg}	ms		0.142	0.081	0.030	
$EI_{avg\%}$	%		2.9	4.1	2.9	
EI_{rms}	ms		0.984	0.608	0.371	
$EI_{rms\%}$	%		20.0	30.7	34.7	
EA_{rms}	ms		0.627	0.480	0.307	
$EA_{rms\%}$	%		12.7	24.2	28.7	

Table 6.4: Detailed results for all Linux-build experiments.

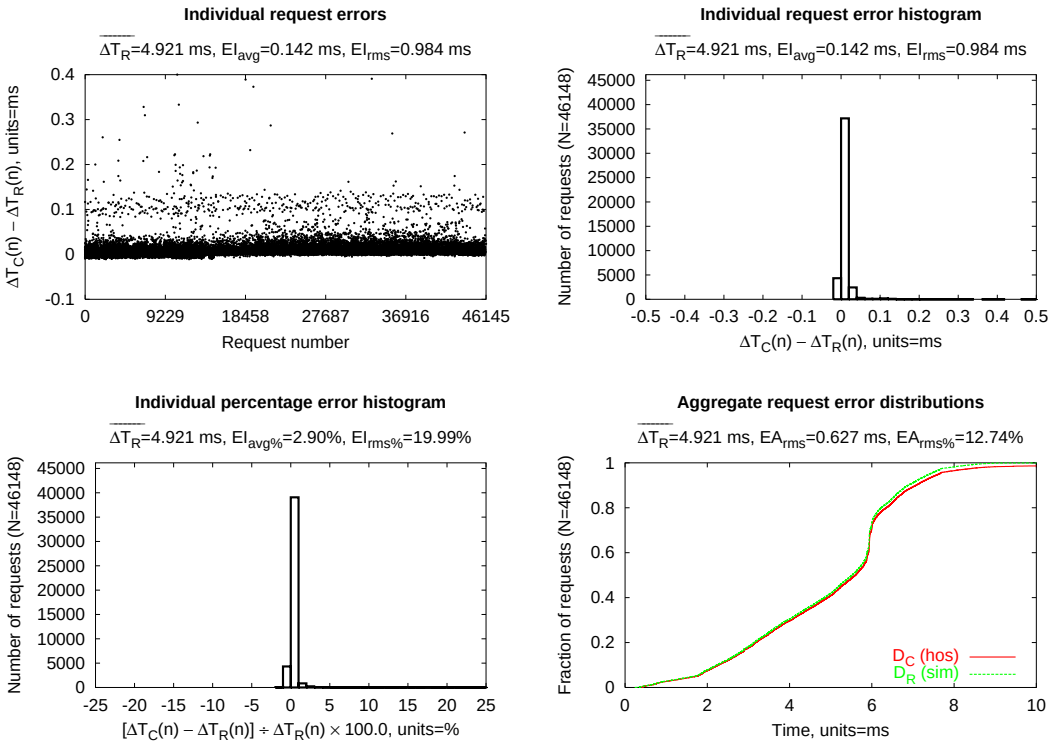


Figure 6.8: Linux-build: Individual results for the Emulated Cheetah model.

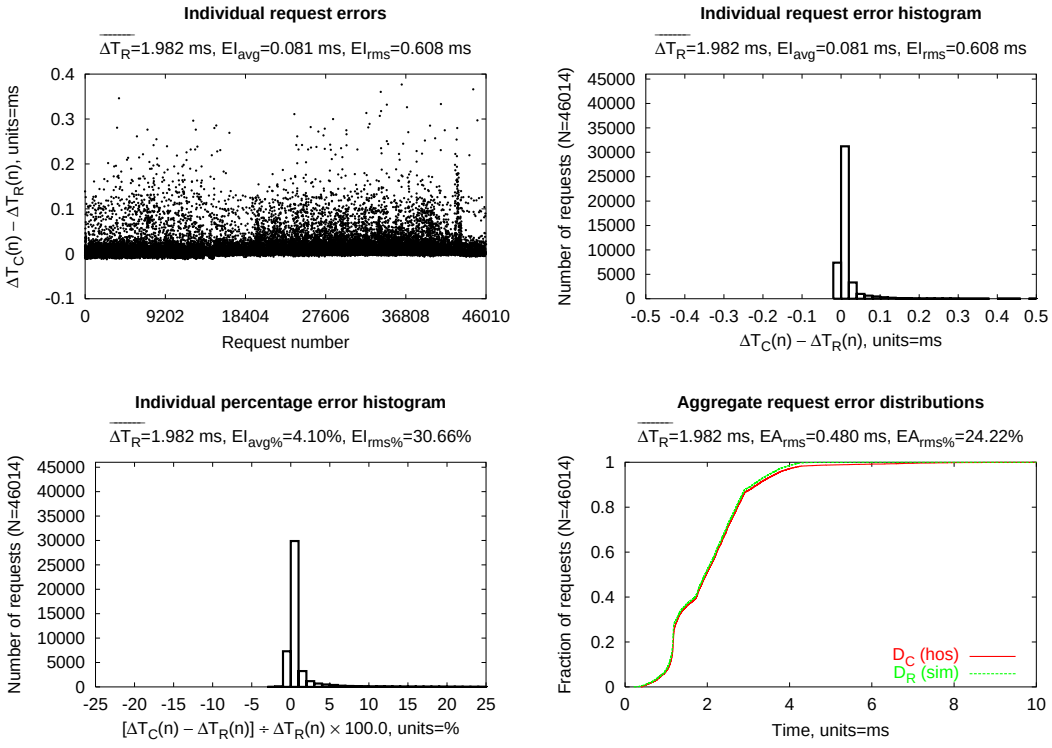


Figure 6.9: Linux-build: Individual results for the Emulated 50K RPM model.

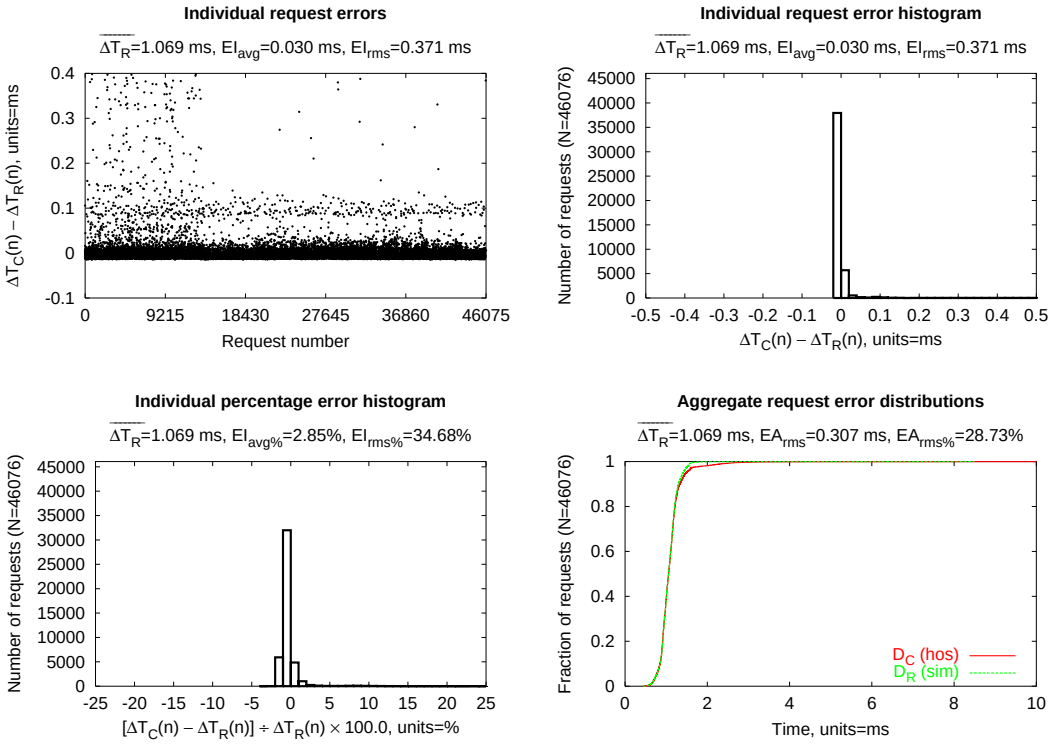


Figure 6.10: Linux-build: Individual results for the Emulated MEMS model.

6.3 Comparison of an emulated disk model against the real disk

As an additional demonstration of the ability of a timing-accurate storage emulator to correctly represent the behavior of modeled devices, we compare the system performance when using a real disk (Real Cheetah) to the system performance when using an emulator configured with a validated model of that disk (Emulated Cheetah). The aggregate results comparing the mean service times achieved using the real disk and emulated disk are presented in Table 6.1 (page 97). Of note is that overall the Emulated Cheetah performed within 0.5% of the Real Cheetah for the total run times of each experiment. The worst measured difference was a 2.2% slowdown obtained during the unpack phase of the SSH benchmark, which itself was only 0.4% slower across all three phases.

We additionally calculate the EA_{rms} aggregate comparison metric between the timestamps obtained during the disk and emulator experiments. These values are not technically the same as the other EA_{rms} values reported elsewhere in this dissertation (nor are they the same as the demerit values reported elsewhere in the literature) as the two runs did not use identical request streams when generating the graph. However, excellent agreement was obtained between the experiments, as shown in the accompanying distributions:

- The aggregate distribution for the PostMark benchmark Figure 6.11.
- The aggregate distribution for the SSH-build benchmark Figure 6.12.
- The aggregate distribution for the Linux build Figure 6.13.

Note that the reference distribution (D_R) is obtained from the disk times for these figure, unlike in the previous graphs, where D_R is obtained from the times reported by the physical device model.

In general our emulated model slightly underpredicts service times for the disk. We believe this is primarily caused by the emulator not fully accounting for the times relating to bus management such as the controller overheads in both bus adapters, as mentioned in Section 5.2.2, the interconnection bus is included in the scope of which components are emulated in these experiments.

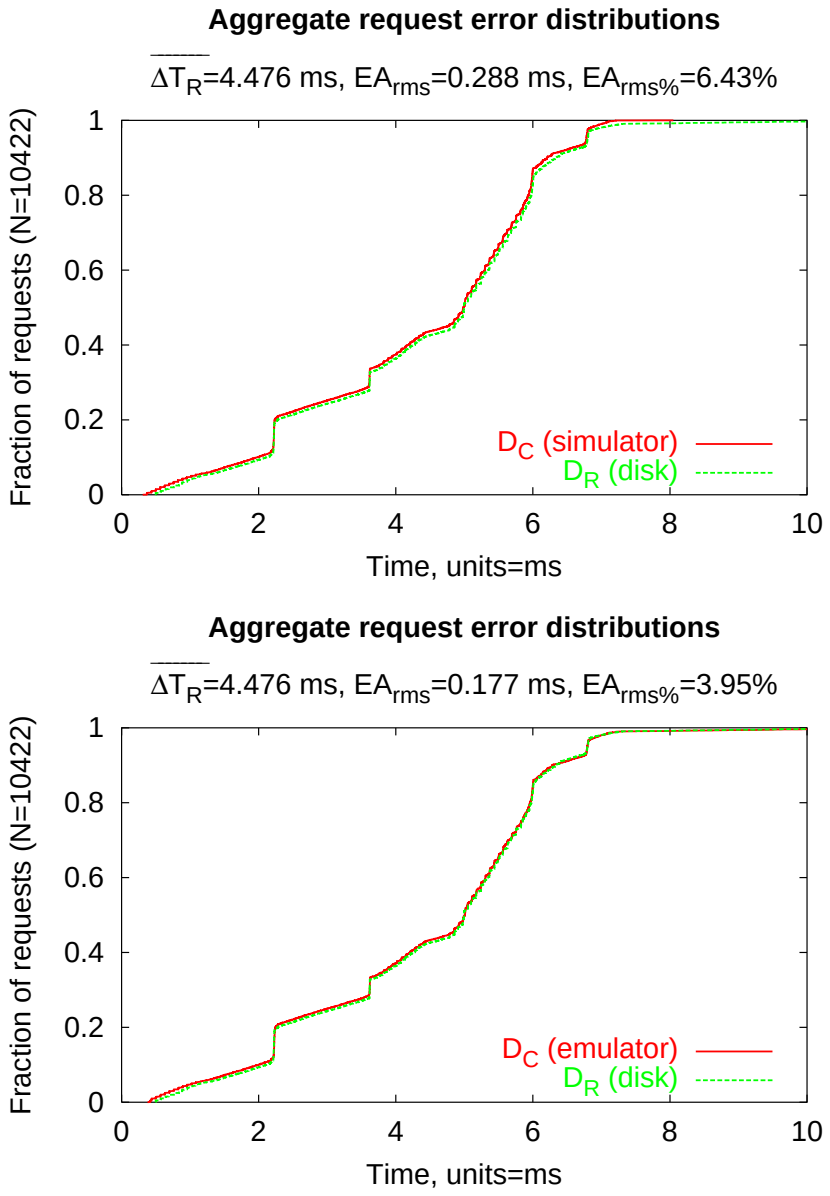


Figure 6.11: PostMark: performance comparison of an emulated disk model and a real disk. *These are comparisons of measurements taken at measurement point MP_1. In the upper graph, the aggregate comparison is made between on one hand the times computed by the simulator containing the disk model, and on the other hand the times measured when using a real disk. In the lower graph, the comparisons are between on one hand the times measured using the emulated device, and on the other hand the times measured when using the real disk.*

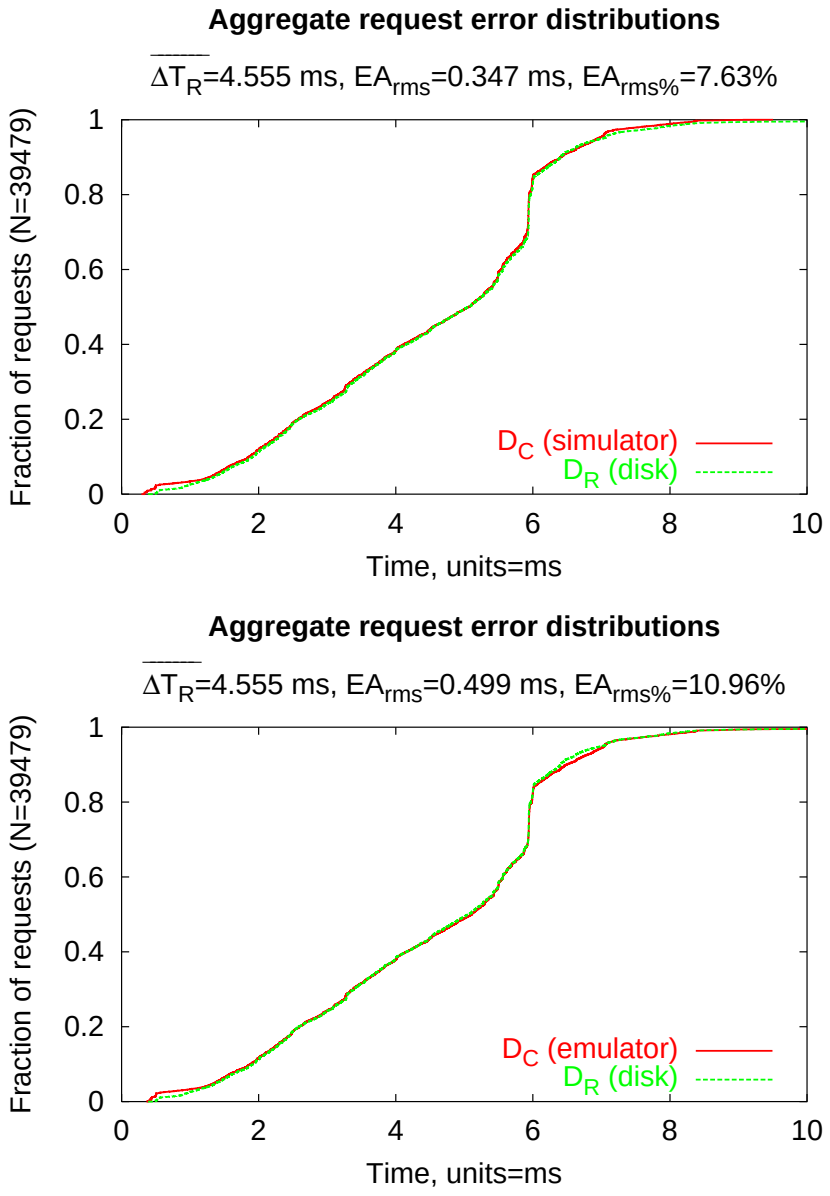


Figure 6.12: SSH-build: performance comparison of an emulated disk model and a real disk.

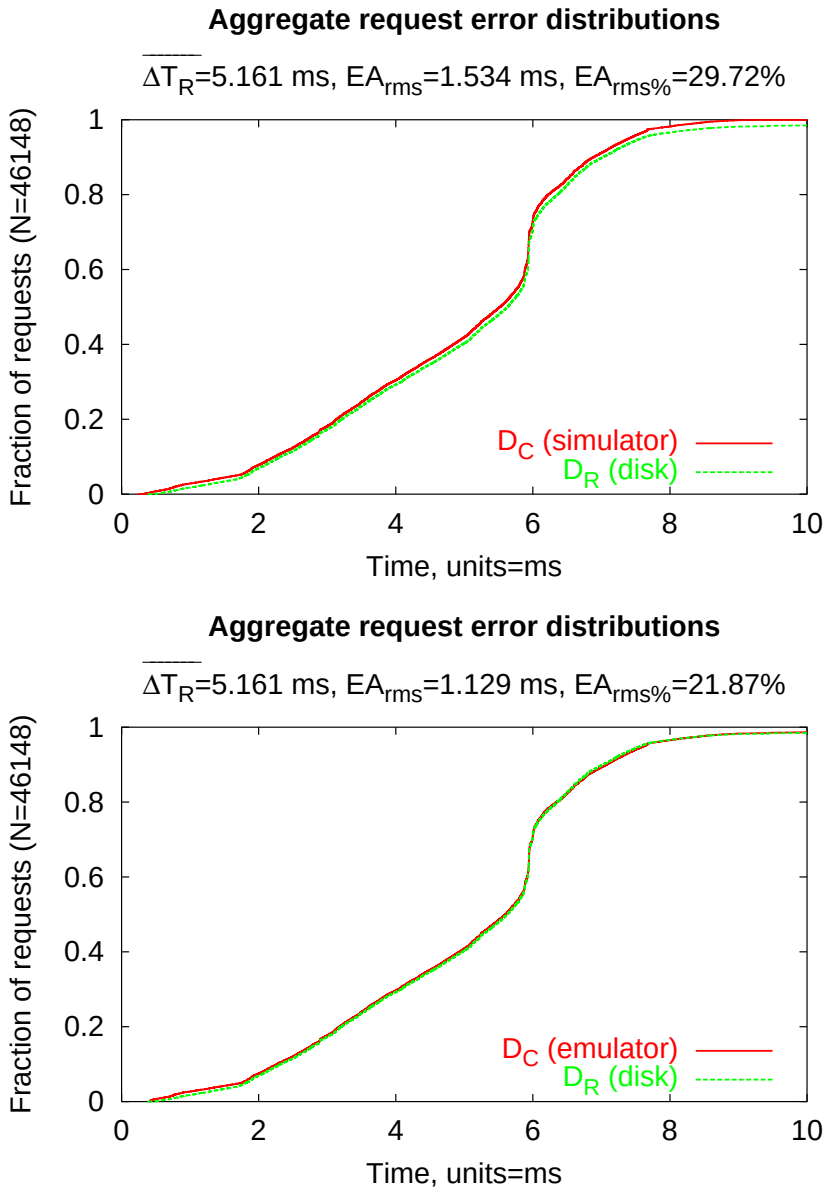


Figure 6.13: Linux-build: performance comparison of an emulated disk model and a real disk.

6.4 Summary of this chapter

A timing-accurate storage emulator can be used for running experiments with real applications on real hardware and real operating systems. We used our implementation of a timing-accurate storage emulator to experiment successfully with three applications on three emulated device models. Further, using an emulated disk model we achieved application run times that were within 0.5% of the run times obtained when using a real disk.

The following chapter demonstrates a strength of timing-accurate storage emulation in evaluating system architectures containing new interfaces to storage devices with autonomous processing capabilities.

CHAPTER 7
INVESTIGATING HYPOTHETICAL INTERFACES TO STORAGE

The previous two chapters demonstrate that it is possible to calibrate and use a timing-accurate storage emulator for full-system experimentation with hypothetical or unavailable storage components. This chapter highlights another strength of timing-accurate storage emulation: the ability to modify both the host operating system and the emulated storage device in order to evaluate new system architectures containing storage devices with new standalone functionality.

This chapter presents a case study of investigating the design and implementation of a disk-based intrusion detection system using the principles of timing-accurate storage emulation. Storage-based intrusion detection systems (IDSes) can watch for and notify administrators about suspicious software executing on a host computer, including many common intrusion toolkits. This chapter argues the advantages of implementing IDS functionality in the firmware of workstations' locally attached disks, where the bulk of critical system files typically reside. To evaluate the feasibility of this approach, we built a prototype disk-based IDS using a timing-accurate SCSI disk emulator. Experimental results from this prototype indicate that it would indeed be feasible, in terms of increased processor and memory requirements, to include IDS functionality in low-cost desktop disk drives.

7.1 Introduction to storage-based intrusion detection

Intrusion detection systems (IDSes) are important tools for identifying malicious activity in computer systems. IDSes are commonly deployed both on end-user computers (the host systems) and at points in the network leading to the host systems. However, there are no perfect intrusion detection systems. All are susceptible to false positives and undetected intrusions. Also, most are much better at detecting probes and attempts to intrude than they are at detecting misbehavior after an intrusion. For example, IDSes running on the same physical hardware as the host system are vulnerable to being turned off when the host is compromised. IDSes running inside network components detect attempts to locate and breach vulnerable systems, but their effectiveness at identifying intruder actions dwindles because many fewer signs are visible once penetration of the host system is achieved.

A storage-based IDS runs inside of a storage device, watching the sequence of requests for signs of intrusions. Storage-based IDSes are a new vantage point for intrusion detection, offering complementary views into system activity, particularly after a successful intrusion has begun. Many root-kits and intruders manipulate files, which can be observed by the disk that stores them. For example, an intruder may overwrite system binaries or alter log files in an attempt to hide evidence of the intrusion. Other examples include adding back doors, Trojan horses, or discreet repositories of intruder content (such as pirated videos). A storage-based IDS can detect such actions. Further,

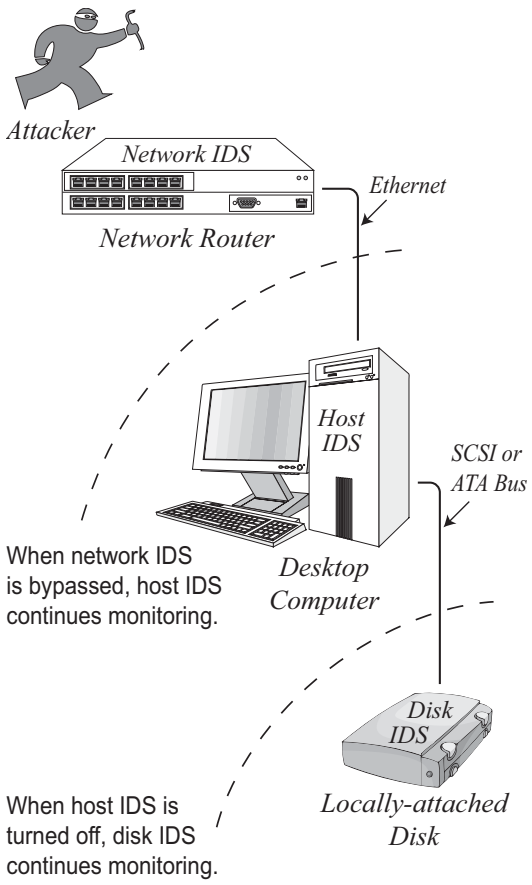


Figure 7.1: The role of a disk-based intrusion detection system (IDS). *A disk-based IDS watches over all data and executable files that are persistently written to local storage, monitoring for suspicious activity that might indicate an intrusion on the host computer. In this example, all three IDSes are administered by an external administrative machine (not shown), perhaps as part of a larger-scale managed computing environment.*

because a storage-based IDS is not controlled by the host operating system (or by the host's IDS), it will continue to operate even when the network-based IDS is circumvented and the host-based IDS is turned off.

The vast majority of systems—and those which are most vulnerable to attack—are single-user workstations with local disks. To be effective in practice, storage-based IDSes must run on the local disk of each workstation. Figure 7.1 shows an example of such a disk-based IDS deployment. The host computer is a standard user desktop and is therefore vulnerable to user mistakes and software vulnerabilities. The storage-based IDS backs up the host and network IDSes, running on a SCSI or IDE/ATA disk with slightly expanded processing capabilities.

A workstation disk with a block-based interface is a challenging environment for intrusion detection. To function, a disk-based IDS must have some semantic knowledge about the file systems contained on the disk, so it can analyze low-level "read block" and "write block" requests for ac-

tions indicating suspicious accesses or modifications to the file system. At the same time, disks are embedded devices with cost-constrained processing and memory resources and little tolerance for performance degradation. Additionally, effective operation will require a secure channel for communication between the administrator and the disk-based IDS; for practical purposes, this channel should seamlessly integrate with the disk's existing block interface.

Much of the design of a disk-based IDS relies on new interactions between the host system or external administrator and the disk's firmware. For example, the administrator will need to send the access policies and other configuration instructions to the disk. In turn, the disk will need to transmit status updates and alerts to the administrator, while at the same time providing ordinary read and write services to the host system.

A practical disk-based IDS product—i.e., one whose cost and deployment complexity are similar in scope to current product offerings—will likely not have a dedicated external network interface for such out-of-band traffic to arrive at or be sent by the disk. Rather, it is likely that such traffic will be routed over the existing storage interconnect between the host system and the device. A timing-accurate storage emulator permits construction of a system with this message-passing characteristic, with the expectation that the results will be more representative of the effects experienced by ultimately deployed device. For example, under emulation-based evaluation the storage bus will actually experience any delays of the ordinary request traffic that are competitively caused by the introduction of new administrative traffic across the bus.

This chapter demonstrates the feasibility of performing intrusion detection inside locally-attached workstation disks. It describes a prototype disk-based IDS built on top of a storage device emulator. This prototype connects to the SCSI bus in a real computer system, looking and "feeling" just like a disk, and monitors real storage requests for behavior that matches the behavior of real-world intrusion tools. Experiments with this prototype demonstrate that the CPU and memory costs will be well within tolerable bounds for modern desktop disks for reasonable rule-sets and workloads. For example, only a few thousand CPU cycles per I/O are needed for over 99% of disk I/Os, and less than a megabyte is needed even for aggressive rule-sets. Moreover, any disk in which the IDS is not enabled would incur no CPU or memory cost, making it feasible to include the support in all devices and enable it only for those that pay for licenses; this model has worked well for 3Com Corporation's network interface card (NIC)-based firewall product [1].

The experiments in this section intentionally do not use the techniques described previously for executing a device simulation model in real time. Instead, the timings (and data) for requests in our experiments are provided by a real disk controlled by the emulation software. Our intention is to examine and demonstrate the usefulness of the extended interface to realistic storage devices that is enabled by the environment of timing-accurate storage emulation. In future experimentation, it is certainly feasible to perform this sort of experimentation using a much broader array of emulated storage device types than the specific disk used here—for example, disks with greater or fewer RAM resources, or small disk arrays.

This chapter motivates workstation disks as an untapped location for real-time intrusion detection (Section 7.2), discusses IDS design challenges relating to the workstation disk environment (Section 7.3, page 120), and describes our prototype disk-based IDS system (Section 7.4) (page 123). It additionally evaluates the prototype's performance (Section 7.5, page 128) and discusses the feasibility of real-world IDS integration (Section 7.6, page 133).

7.2 Background and motivation

This section motivates the use of workstation disks as a target environment for intrusion detection, discusses previous work that demonstrates the effectiveness of a storage-based IDS in identifying real-world intruder behavior, and reviews work related to storage-based IDSes.

7.2.1 Intrusion detection in storage

Storage-based intrusion detection enables devices such as workstation disks to watch storage traffic for suspicious behavior. Because storage-based IDSes run on separate hardware with a limited external interface, they enjoy a natural compromise-independence from the hosts to which they are attached. As shown in Figure 7.1, an attacker who breaches the security of the host must then breach a separate perimeter to disable a security system on a storage device. Also, because storage devices see all persistent activity on a computer, several common intruder actions [44, p. 218][138, pp. 363–365] are quite visible inside the storage device.

A storage IDS shines when other IDSes have been bypassed or disabled but neither the storage device nor the administrator's computer have been compromised. Our design addresses a threat model where an attacker has full software control but not full hardware control over the host system. This could come in the form of an intruder breaking into a host over the network, or from a user with administrative privileges mistakenly executing malicious software. We do not explicitly protect against an aggressive insider with physical access to the disk; other researchers continue to investigate hardware-based solutions to preserving the security-related semantic integrity of a device's physical external boundary.

Administrators can watch for a variety of possible intruder actions inside a storage-based IDS. First, the IDS could watch for unexpected modifications to file data or meta-data; this is similar to the functionality of the Tripwire utility [99], with the key difference being that a storage-based IDS both monitors every access to the disk and performs this monitoring in real time. The IDS could also watch for suspicious access patterns, such as a non-append write to a system log file—a system log should remain persistent and only increase in size with no changes to already written data, and a common intruder action is to selectively erase log entries to nonsuspiciously eliminate evidence of a successful intrusion. Additionally, the IDS could watch for loss of file integrity for well-structured files such as `/etc/passwd` or database files. And it could watch for suspicious content, such as malformed file names or known virus signatures.

However, despite its strengths and advantages, storage-based intrusion detection is not a panacea for monitoring and protecting computer systems against intrusions. While false positives should be infrequent, legitimate actions that modify a watched file will create an alert to the administrator. The broader the scope of watched actions, the higher the frequency of false positives will be. On the other hand, a storage IDS could miss an intrusion entirely if it is configured to watch too limited a set of files, so there is a tension for the administrator to balance these issues when identifying the correct scope and types of rules for a disk-based IDS. Additionally, including IDS functionality inside the storage device will likely have an ultimate performance impact on the host computer's workload, or alternatively will require an increase in the device's cost to mitigate any performance degradation.

7.2.2 Real-world efficacy of a storage IDS

During our initial investigation into the applicability of storage-based intrusion detection, we analyzed an NFS server that had been extended to include IDS functionality [125]. Our analysis demonstrates that storage-based intrusion detection is indeed an effective tool for detecting the effects of real-world attacks, and that the NFS-based implementation is efficient in its processing and memory requirements.

To evaluate the real-world efficacy of a storage-based IDS, Adam Pennington analyzed the behavior of eighteen publicly-available intrusion tools designed to be run on compromised systems. This analysis indicated that 83% of the intrusion tools modify one or more system files, and that these modifications would be noticed by their storage-based IDS when it monitored a simple, default rule-set. Also, 39% of the tools alter a system log file, an action which also would be detected by a storage-based IDS using a default rule-set. Later, we supplemented his analysis with a forensic analysis of a real desktop computer that had been unexpectedly compromised.[7.1] This forensic analysis concluded that a storage-based IDS would have spotted the intrusion because of various system binaries that were modified during the attack.

As a supplement to our previous work we analyzed some brief traces of a desktop workstation's disk and reported preliminary results indicating that false positives would very rarely be reported by a storage-based IDS, with the exception of (planned) nightly rewrites of the password file. Section 7.5.4.1 presents results from a more extensive collection of traces, confirming and underscoring this result.

[7.1] The tools used during the intrusion were preserved for later forensic analysis using the rather abrupt method of "yanking out the power cord" after quickly saving the active process list and issuing the `sync` command to save the operating system's write buffers. This action prevented the intrusion script from erasing the related files and log entries before the analysis could proceed. The attack was discovered because it targeted a small server used as a private network gateway, whose older disk made a loud clicking noise after every seek. After exploiting a vulnerability to gain administrative access, the intruder began searching all files on the disk for credit card numbers and passwords. This search resulted in a torrent of seek-related noise just as the owner sat down to eat a bowl of cereal nearby. It is fortunate that the owner's cereal was soggy that morning, otherwise he might not have noticed anything over the sound of his own crunching!

7.2.3 Related work to storage-based intrusion detection

Intrusion detection is a well-studied field. One of the earliest formalizations of intrusion detection is presented by Denning et al. [43]. Tripwire is one of the more well-known intrusion detection systems [99, 100]; we use the suggested Tripwire configuration as part of the basis for the rule-set in the experiments in this chapter. Investigations have been made into intrusion detection systems founded on machine learning [52] as well as static rules to watch system calls on a machine [102]. In 1998, Axelsson surveyed the state-of-the-art for intrusion detection systems [12].

Recent research has explored other ways of utilizing similar protection boundaries to the device interfaces exploited in this chapter. Chen and Noble [36] and Garfinkel and Rosenblum [65] propose using a virtual machine monitor (VMM) that can inspect and observe machine state while remaining compromise-independent of most host software. This could be expanded by adding a storage-based IDS into the virtual machine's storage module. Additionally, recent work explores the idea of hardware support for doing intrusion detection inside systems [61, 182].

Adding IDS functionality to storage devices can be viewed as part of a recurring theme of migration of processing capability to peripheral devices. For example, several research groups have explored the performance benefits of offloading scan and other database-like primitives to storage devices [3, 98, 131]. Other research has explored the use of device intelligence for eager writing [34, 171]. Recently, Sivathanu et al. proposed the general notion of having storage devices understand host-level data semantics and use that knowledge to build a variety of performance, availability, and security features into "semantically-smart" disk systems [156]. A disk-based IDS is a specific instance of such a semantically-smart system.

7.3 Design issues for disk-based intrusion detection systems

There are four main design issues for a storage-based intrusion detection system: specifying access policies, securely administering the IDS, monitoring storage activity for policy violations, and responding to policy violations. This section discusses the aspects of these that specifically relate to the challenging environment of workstation disks, and how the technique of timing-accurate storage emulation supports the evaluation of these design issues.

7.3.1 Specifying access policies

For the sake of of usability and correctness, there must be a simple and straightforward syntax for human administrators to state access policies to a disk-based IDS. Although a workstation disk operates using a block-based interface, it is imperative that the administrator be able to refer to higher-level file system objects contained on the disk when stating policies. As an example, an appropriate statement might be: *Warn me if anything changes in the directory /sbin.* In our experience, Tripwire-like rules [99] work well for specifying policies to a disk-based IDS.

A disk-based IDS must be capable of mapping such high-level statements into a set of violating interface actions. This set of violating actions may include writes, reads (e.g., of honeytokens [158] such as `creditcards.txt`), and interface-specific commands (such as the FORMAT UNIT command for SCSI). One such mapping for the above "no-change" rule for `/sbin` could be: *Generate the alert "the file /sbin/fsck was modified" when a write to block #280 causes the contents of block #280 to change.* To accomplish this mapping, the IDS must be able to read and interpret the on-disk structures used by the file system. However, the passive nature of intrusion detection means that it is not necessary for a disk-based IDS to also be able to modify the file system, which greatly simplifies implementation.

Workstation disks are frequently powered down. These mappings must be restored to the IDS (i.e., from an on-disk copy) before regular operation commences. We anticipate this requiring perhaps a few megabytes of private disk space. This approach is no different from other tables kept by disk firmware, such as those for tracking defective sectors and predicting service times efficiently.

A disk-based IDS is capable of watching for writes to free space that do not correspond with updates to the file system's block allocation table. Storing object and data files in unallocated disk blocks is one method used by attackers to hide evidence of a successful intrusion [75]. Such hidden files are difficult to detect, but are accessible by processes (such as an IRC or FTP server) initiated by the attacker. Depending on the file system, watching free space may cause extra alerts to be generated for short-lived files which are deleted after the contents are written to disk but before the allocation table is updated.

7.3.2 Disk-based IDS administration

For effective real-time operation of a disk-based IDS, there must be a secure method for communication between an administrator's computer and the IDS. This communication includes transmitting access policies to the IDS, receiving alerts generated by the IDS, and acknowledging the receipt of policies and alerts. Unlike in a server-based environment, a workstation disk will likely not have direct access to a communications network to handle such administrative traffic. Instead, this traffic must be routed through the host and over the existing physical link connecting the host with the disk. In other words, the host must be actively involved in connecting the administrator's console and the disk, playing a role much like a switch in the network path.

This communications model presents several challenges. For one, in most block-based storage interconnects, the disk takes the role of a passive target device and is unable to initiate a data transfer to the host. Instead, the host must periodically poll the IDS to query if any alerts have been generated. Also, the administrative communication path is vulnerable to disabling by an attacker who compromises the host system. In response to such disabling, both the administrator and the IDS could treat a persistent loss of communication as a policy violation. The IDS could alter its behavior as described in Section 7.3.4, while additionally logging any subsequent alerts for later transmission to the administrator.

The communications channel connecting the disk-based IDS with the administrator must be protected both from eavesdropping and tampering by an attacker. Such a secure channel can be implemented using standard cryptographic techniques. For the network intruder and rogue software concerns in our security model, it is sufficient for secret keys to be kept in the disk's firmware. If physical attacks are an issue, secure coprocessors can be used on the disk [182]. Such additions are not unreasonable for future system components [69, 109].

7.3.3 Monitoring for policy violations

Once the administrative policy is received by a disk-based IDS, all storage requests arriving at the disk should be checked against the set of violating interface actions. A check should be performed for every block in a request: a write to block 72 of length 8 blocks should check for violating actions on any of blocks 72–79. As this check is in the critical path of every request, it should be implemented as efficiently as possible.

Some file system objects, such as large directories, will span multiple sequential disk blocks. Space for such objects will generally be allocated by the file system in a multiple of the file system block (hereafter, *fs-block*) size. We found it convenient for our disk-based IDS prototype to evaluate incoming requests by their impact on entire fs-blocks instead of on individual disk blocks: *Generate the alert "the file /sbin/fsck was modified" when a write to fs-block #35 causes the contents of fs-block #35 to change.*

When authorizing a write request, a disk-based IDS may need to first fetch the previously written (old) data for that block from the disk. The old data can then be compared with the new data in the write request to determine which bytes, if any, are actually changed by the write. Such a check would also be necessary when a block contains more than one file system object—for example, a single block could contain several file fragments or inode structures—and different administrative policies apply to the different objects. A similar check allows the system to quell alerts that might otherwise be generated during file system defragmentation or reorganization operations; no alerts should be generated unless the actual contents or attributes of watched files are modified.

7.3.4 Responding to policy violations

For an intrusion detection system in a workstation disk, the default response to a confirmed policy violation should be to prepare an administrative alert while allowing the request to complete. This is because the operating system may halt its forward progress when a locally-attached disk returns an error or fails to complete a request, especially at boot time or during crash recovery.

There are several other possible responses for a storage-based IDS after a policy violation [125]. The most relevant of these for a disk-based IDS include artificially slowing storage requests while waiting for an administrative response, and internally versioning all subsequent modifications to aid the administrator in post-intrusion analysis and recovery [160]. To support versioning without needing to modify the file system, a disk-based IDS can copy-on-write the previous contents of

blocks to a private area on the disk that is inaccessible from the host computer. The administrator can later read the contents of this area over the administrative communication channel to assist in intrusion recovery or analysis.

After a policy violation, the IDS should make note of any resultant changes to the on-disk file system structure and adjust its behavior accordingly. For example, for the example policy specified in Section 7.3.1 (*Warn me if anything changes in the directory /sbin*), when a new file is created in the `/sbin` directory the system should first generate an alert about the file creation and then begin monitoring the new file for changes.

7.4 Prototype implementation

We built a prototype disk-based IDS called the *IDD* (Intrusion Detection for Disks). The IDD takes the form of a PC masquerading as a SCSI disk, enhanced with storage-based intrusion detection. From the perspective of the host computer, the IDD looks and behaves like an actual disk with this additional IDS functionality, accepting ordinary storage traffic and administrative commands from the host system, and providing administrative responses and replying to requests at the appropriate time. This section describes architectural and implementation details of this prototype.

7.4.1 Architecture

Figure 7.2 shows the high-level interaction between the user's desktop or notebook (host) computer and the IDD components. These components are located both in the host's locally-attached disk (the timing-accurate storage emulator) and in the Administrator's Console, which is connected to the host over a network. Although not shown, the administrator's console would likely also serve as a central point of control for other intrusion detection systems running on other users' host computers.

There are two primary functions of this architecture. The first function, *storage traffic monitoring*, is implemented by the Policy Monitor and the IDD Block Cache. This component handles the mapping of administrative policy into violating interface actions. It also monitors all ordinary storage traffic for real-time violations and generates alerts. The second function, *administrative communication*, is implemented by the Administrative Bridge process (which runs in the host being watched) and the Request Demultiplexer. The bridge process forwards commands from the administrator to the IDD and conveys administrative alerts from the IDD to the administrator. The request demultiplexer identifies which incoming SCSI requests contain administrative data and handles the receipt of administrative policy and the transmission of alerts.

7.4.2 Storage traffic monitoring

The policy monitor bridges the semantic gap between the administrator's policy statements and violating SCSI requests, and it audits all storage traffic from the host computer in real time. The policy monitor operates at the file system block (fs-block) level, as discussed in Section 7.3.3.

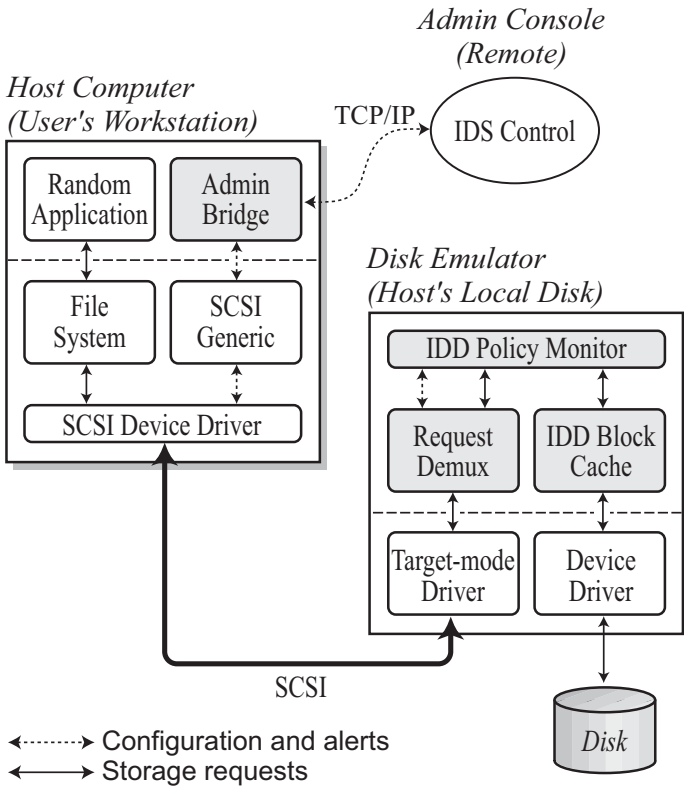


Figure 7.2: Intrusion Detection for Disks (IDD) prototype architecture. *This figure shows the communications flow between a user's host computer and the IDD components, including the IDS-enabled disk and the administrator's console machine. The shaded boxes implement the primary IDD functions of storage traffic monitoring and administrative communication (see Section 7.4.1). Ordinary storage traffic is initiated by application processes, passed across the SCSI bus, checked by the policy monitor, and finally serviced by the disk. Administrative traffic is initiated by the administrator, passed across a TCP/IP network, received by the bridge process on the host computer, passed across the SCSI bus, and finally serviced by IDD's policy monitor.*

Request block numbers are converted to the relevant partition and fs-block numbers upon request arrival. (Hereafter in this section, "block" refers to a fs-block.)

The IDD has a relatively simple semantically-smart understanding of the on-disk file system structures. The IDD currently understands the Second Extended (ext2) file system used by Linux-based host computers [27]; to support this we hard-coded the structure of on-disk meta-data into the policy manager. For ext2 this included the ext2 superblock, inode, indirect block, and directory entry structures. As an alternative to hard-coding, we envision a more flexible administrative capability where the relevant file system details could be downloaded to the disk across an extended storage interface. Both of the above approaches are feasible under an economic model of cooperation between host OS vendors and disk manufacturers. During initial configuration, the administrator would specify the FS used by the host computer. It may also be possible for the disk to use grey-box techniques to acquire these details [156].

As administrative policy is specified, the IDD receives a list of files whose contents should be watched in particular ways (e.g., for any change, for reads, or for non-append updates). For each of these watched files, the IDD traverses the on-disk directory structure to determine which meta-data and data blocks are associated with the file. These blocks are labeled "watched blocks" and are added to a table of watched blocks, described below. Each such block is then associated with an number of access check functions that evaluate whether a block access violates a given rule.

The mapping from file-system-level policy to potential block access violations must be thorough. Consider the example rule *Warn me if the file /bin/netstat in partition 2 changes*. For ext2, the mapping expands to individual access check functions associated with: (1) the second entry in the disk's partition table, (2) certain fields in the ext2 super block, (3) fields in the root inode, (4) indirect blocks—including doubly- and triply-indirect—for the root inode's directory entries, (5) the inode number associated with `bin` in the root directory, (6) fields in the `bin` inode, (7) block pointers for `bin`'s directory entries, (8) the inode number for `netstat`, (9) fields in the `netstat` inode, (10) indirect blocks for `netstat`, and (11) each of the data blocks for `netstat`. Newer file systems that use complicated internal indexing structures, such as NTFS and ReiserFS, may involve further steps to fully cover which changes impact a rule.

7.4.2.1 The watched block table (WBT)

The watched block table (WBT) is the primary structure used by the policy manager for storage request monitoring. It is stored in private, reserved disk space for persistence and paging. A WBT entry contains a monitored block number, a list of the associated access check functions for the block (see below), and a human-understandable filename (e.g., */bin/netstat*) to be sent whenever the ACF reports a violation. The IDD maintains a separate WBT for each monitored partition, as well as a WBT for the partition table and other unpartitioned disk space. For efficiency, our implementation uses a B*-tree to maintain lists of monitored block extents.

When new storage requests arrive from the host computer, the IDD checks whether any of the request's blocks appear in the internal WBT. In the expected very common case, no matches are found. This means there are no possible policy violations caused by the request, so no further IDS-related processing is required. (The other case is discussed in Section 7.4.2.2.) It is imperative that this WBT lookup be efficient, as it must be performed in the critical path (before the media transfer) for every write request. It cannot proceed in parallel with the media transfer for writes, because in the case of a WBT match the IDD may need to fetch the old data for comparison purposes.

It may be desirable to minimize the memory footprint used by a disk-based IDS, to conserve disk memory for traditional data buffering. To achieve this, portions of the WBT could be paged. The list of monitored block numbers must always remain in-core, but the remaining information (the access check function pointer and alert-time explanation) could be demand paged once a monitored block is accessed.

7.4.2.2 Access check functions (ACFs)

If one or more of the blocks of a request appear in the WBT, the IDD must perform extra processing to determine whether the request actually causes a rule violation. This is necessary because multiple file system objects can appear in a single block (e.g., directory entries), only some of which may set off rules when updated. Unlike the common-case analysis above, it is not necessarily paramount that these checks have no performance impact on the host's request stream: since an update to a watched block may indicate an intruder action, it is imperative for the disk to determine whether the specified rules have been violated.

As discussed in Section 7.3.3, checking the validity of a write request may require the old data to be read from the disk. We call this process an "interposed read" (IR). Blocks that have a high probability of being fetched via IR, such as those containing monitored inodes and directory entries, can be cached internally by the IDD to reduce IDS-related delays. Note that for the reasons above, this cache is primarily designed to quickly execute ACFs on block updates that will ultimately not generate an alert.

As examples from our ext2-based implementation, the ACF for a data block and the "any change" policy would simply compare the old contents of the block with the new. The ACF for a directory entry block and the "any change" policy would check the old and new contents to determine only if a particular filename-to-inode-number mapping changed (e.g., `mv file1 file2` when `file2` is watched) and, if so, whether the new inode's file contents match the old inode's file contents. The ACF for an inode block and the "non-append updates" policy would compare the old and new inode contents to ensure that the access time field and the file size field only increased. Complicating the logic for this rule, the last allocated block pointer is allowed to change, but only if none of the allocated bytes (i.e., bytes numbered less than the file size) change between the old data block and the new.

7.4.3 Alert generation and communication

The administrative communications channel is implemented jointly by the bridge process and the request demultiplexer. The administrator sends its traffic directly to the bridge process over a TCP/IP-based network connection. The bridge process immediately repackages that traffic in the form of specially-marked SCSI requests and sends those across the SCSI bus. When these marked requests arrive inside the IDD, they are identified and intercepted by the request demultiplexer. The endpoints of the secure channel are the administrator's computer and the request demultiplexer.

The repackaging in the bridge process takes different forms depending on whether the administrator is sending new policies (outgoing traffic) or polling for new alerts (incoming traffic). For outgoing traffic, the bridge creates a single SCSI WRITE request containing the entire message. The request is marked as containing administrative data by setting an unused flag in the SCSI command descriptor block. The request is then sent to the bus using the Linux SCSI Generic passthrough driver interface.

For incoming traffic, the bridge creates either one or two SCSI READ requests. The first request is always of fixed-size (we used 8 KB) and is used to determine the number of bytes of alert data waiting in the IDD to be fetched: The first 32 bits received from the IDD indicate the integer number of pending bytes. The remaining space in the first request is filled with waiting data. If there is more data waiting than fits in the first request, a second request immediately follows. This second request is of appropriate size to fetch all the remaining data. These requests are marked as containing administrative data and sent in the same manner as for outgoing traffic. Once the bridge has fetched all the waiting data, it forwards the data to the administrator over the network.

The administrative communication channel must also be reliable in the face of message duplication or omission due to network problems or malicious attack. Our implementation uses per-message sequence and acknowledgement numbers to ensure that such errors are detected. The IDD administrator sends one pair of messages (outgoing and incoming) per second by default. In order to reduce administrator-perceived lag, this frequency is temporarily increased whenever recent messages contained policies or alerts.

7.4.4 Alternative architectures

We intentionally chose a prototype architecture that represents a "limit study" of including comprehensive processing capabilities inside the device, and making it independently responsible for interpreting the on-disk data structures and executing its responses to perceived intrusions. Other architectures are also practical and are perhaps more feasible than our prototype architecture. For example, the scope of the policy monitor can be greatly simplified—the mapping of administrative policy to disk blocks can occur instead inside the administrative console, which has far more powerful computational and memory resources than a disk, and simply downloaded in bulk to the disk-based IDS. This would require that the administrative console be able to read data from the disk—which itself could be achieved through an additionally extended storage interface, or again more simply with the assistance of a support application process executing on the host system. This would result in a simpler firmware implementation with less dependence on which file system is used on the disk. However, this reduction in complexity would come at the expense of longer and more frequent communications with the administrator and would additionally require more detailed interactions between the disk and administrator whenever rules are violated (by an intruder or otherwise) or are updated by the administrator. Our limit study evaluates the complete system and its capabilities when all necessary resources are available on the disk and, conversely, quantifies what resources are required.

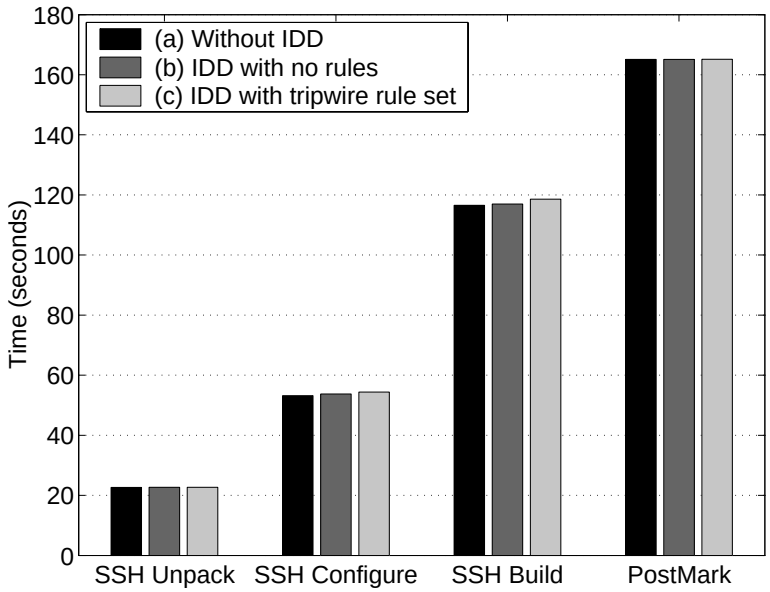


Figure 7.3: Non-alert-generating application benchmarks. *These graphs show the impact of the common-case IDS action (WBT check only) on the SSH-build and PostMark benchmarks. The emulated disk is mounted synchronously in these experiments. Case (a) is the emulated disk with the IDD disabled; case (b) is the emulated disk with the IDD engaged but no rules set; case (c) is the emulated disk with the IDD engaged and the fully enabled ruleset. These results show that there is nearly zero application-level impact in the common case of no updates to watched blocks.*

7.5 Prototype evaluation

This section examines the performance and memory overheads incurred by IDD, our prototype disk-based IDS. Our results indicate that the processing and memory resources required by the disk-based IDS are not unreasonable for inclusion in workstation disks.

7.5.1 Experimental setup

The host and emulator systems were configured as described in Chapter 4, specifically using the QLogic QLA2100 fibre channel adapters[7.2] to connect the systems, with the exception that a Fujitsu Enterprise SCSI-3 model MAN3184MP disk (described in Table 4.1 on page 44) was used to compute the device-related timings and to store written data. In an effort to exercise the worst-case storage performance, the disk emulator was mounted synchronously by the host computer and

[7.2] The use of the QLogic HBA was mandated due to an unfortunate limitation in the FreeBSD target-mode support for Adaptec bus adapters. Specifically, the Adaptec device driver does not currently support tagged command queuing (TCQ) in target-mode. TCQ enables multiple outstanding requests to be sent to the emulation software. As our administrative traffic appears as ordinary storage requests on the bus (with a special flag set), we were unable to send administrative alerts for a request until after the request completed when using the Adaptec driver . The QLogic device driver fully supports TCQ, which enables us to delay violating request completions from being sent until IDD sends an alert and receives post-alert instructions from the administrative console.

caching was turned off inside the backing store disk. As discussed previously, these experiments achieve "timing-accuracy" in a different form than that described in Chapter 5, in that here we rely on a real device to provide the mechanical timings for the emulated IDS-enhanced device.

We do not argue that embedded disk processors will have a 2 GHz clock frequency, as is the case for our emulation system; this is perhaps an order of magnitude larger than one might expect. However, an actual disk-based IDS would be manually tuned to the characteristics of the platform it runs on (e.g., the disk's SCSI ASICs would obviate much of the IDD's communication and interposition overheads) and would therefore run more efficiently than the IDD, perhaps by as much as an order of magnitude. To compensate for this uncertainty, we report processing overheads both in elapsed time and in processor cycle counts, the latter of which provides a reasonably portable estimate of the amount of work performed by the IDD.

To approximate the conditions of a production file system deployed in a real environment, we created a disk image of a freshly installed Red Hat Linux 8.0 desktop system. This image is loaded into the emulator at the start of each experiment, after which the emulated disk is mounted synchronously in a root-level directory on the host computer. For each experiment with the IDD "fully enabled" we set the administrative policy to match the default Tripwire rule set for Red Hat Linux [166].

Our experiments use a microbenchmark and two macrobenchmarks. Our alert-generating microbenchmark cycles 1000 times over a single file operation on each of 1000 files, all of which will generate alerts when the IDD is enabled. We use SSH-build and PostMark as non-alert-generating macrobenchmarks, as described in Section 6.1.2. Each macrobenchmark result represents the average of 10 runs of the test program on a warm system, where the system was warmed by loading the rules into the IDD and running the test program once.

7.5.2 Base resource requirements

Expanding out the default Tripwire rule-set for Red Hat Linux on our freshly installed disk image resulted in rules being set on 32,970 files, covering a total of 447,829 watched blocks, which we coalesce into 69,705 contiguous extents. This rule coverage represents watching 14% of the total number of files in the system.

For our IDD prototype, the fully-enabled rule-set results in a baseline memory footprint of 6,928 KB. This includes 36 KB for the executable image size, 660 KB for internal non-rule-support data structures, and 6,230 KB for internal structures that track rules and the lists of watched blocks. This works out to 200 bytes per file watched. Because of our architectural division between the fast-lookup WBT (required to be in-core) and the non-common-case ACF processing (allowed to be paged to the private disk area), only 37% (2,300 KB) of the total need be in-core during ordinary execution.

When the rule-set is first loaded by the IDD, it makes an initial pass through the quiescent file system to initialize the lists of watched blocks and other structures. In our implementation, this time

	(a)	(b)	(c) data [not cached]	(c) data [cached]	(c) meta-data [not cached]	(c) meta-data [cached]
Bus transfer time	0.263 (0.014) 530,000	0.267 (0.070) 530,000	0.270 (0.004) 540,000	0.273 (0.017) 540,000	0.278 (0.125) 550,000	0.270 (0.005) 540,000
Emulator overheads	0.024 (0.006) 50,000	0.063 (0.020) 130,000	0.236 (0.016) 470,000	0.241 (0.025) 480,000	0.246 (0.351) 490,000	0.165 (0.012) 330,000
WBT lookup time	n/a	n/a	0.003 (0.001) 5,000	0.004 (0.001) 8,000	0.002 (0.001) 4,000	0.002 (0.001) 4,000
ACF time	n/a	n/a	0.030 (0.006) 60,000	0.042 (0.009) 80,000	0.073 (0.008) 150,000	0.101 (0.054) 200,000
IR time or cache lookup	n/a	n/a	2.520 (0.958) 5,000,000	0.135 (0.013) 270,000	3.204 (1.197) 6,380,000	0.001 (0.001) 2,000
Disk service time	5.21 (0.356) 10,442,000	5.190 (0.141) 10,350,000	5.866 (0.029) 11,680,000	3.884 (1.813) 7,740,000	5.822 (0.121) 11,600,000	3.877 (1.740) 7,720,000
Total time	5.50 (0.365) 10,972,000	5.510 (0.268) 10,980,000	8.926 (0.958) 17,790,000	4.580 (1.812) 9,130,000	9.627 (1.286) 19,180,000	4.419 (1.740) 8,790,000

Table 7.1: Alert-generating microbenchmark. *The top row of each cell gives the average and standard deviation for that phase in milliseconds; the bottom row gives the approximate cycle count. This table decomposes the service time of write requests in our alert-generating microbenchmark. In the row headings, WBT refers to the watched block table, ACF to the access check functions, and IR to the interposed reads, as described in Section 7.4.2. Each benchmark is run in three configurations: case (a) is the emulated disk with the IDD disabled; case (b) is the emulated disk with the IDD engaged but no rules set; case (c) is the emulated disk with the IDD engaged and the fully-enabled ruleset. There are four variants of case (c): "data" involves a write into a watched file; "meta-data" represents a change of the modification time in the file's inode, which results in 32 ACF invocations—one for each inode in the inode block. The "cached" and "not cached" cases represent whether all interposed blocks were already cached in the IDD Block Cache. The cases marked "n/a" (not applicable) have no rules set, and therefore incur no WBT lookup, ACF execution, or IR servicing time.*

is dominated by the number of disk I/Os required to load the relevant inode and directory structures. By caching certain the frequently used blocks (such as the superblock, root inode and root directory blocks, as well as recently touched inodes and directory entries) this requires 22,556 disk I/Os and takes a total of 160 seconds, or about 4.9 ms per rule. This initialization isn't necessary every time the disk is powered up: once the rule-set is loaded, the IDD's internal state can be written to disk upon power down and restored quickly when restarted. It may be desirable for a deployed disk-based IDS to periodically repeat this process in the background, just to verify the consistency of its internal state.

7.5.3 Common-case performance

As discussed in Section 7.4.2.1, it is imperative that the WBT lookup be efficient, as it must be performed in the critical path (before the media transfer) for every write request. We examine the impact of the common-case WBT lookup both in the aggregate using the macrobenchmarks and at the individual request level using the microbenchmark.

Our macrobenchmark results are shown in Figure 7.3. SSH-build generates 46,217 disk requests in 335 seconds, and PostMark generates 35,467 requests in 165 seconds. These graphs show both that the overhead of the IDS infrastructure itself can be small [as shown by case (b), 0.01–0.1% for our implementation] and that the WBT lookup time is insignificant [case (c), 0.02–1.3%] compared with the request times. Our microbenchmark results, shown in Table 7.1, show that the total overhead for WBT lookup is about 250 μs. These results show that it is indeed possible to do the requisite common-case IDS processing with no discernible effect on the application-level workload.

7.5.4 Updates to watched blocks

If one or more of the blocks of a request appear in the WBT, the IDD must perform extra processing to determine if the request actually causes a rule violation. Unlike the common-case analysis above, it is not paramount that these checks have no performance impact on the host's request stream: since an update to a watched block may indicate an intruder action, it is imperative for the disk to determine whether the specified rules have been violated. We examine the impact of watched block checking using the alert-generating microbenchmark. The results are shown in Table 7.1.

This analysis assumes that the entire WBT is kept in-core but that none of data blocks are cached by the IDD. Depending on the system state, the actual results could be worse or better. If the WBT were fully paged to disk, the request time would increase by approximately 30% as an additional interposed request would occur to page in the appropriate entry in the WBT. Conversely, if all the relevant data blocks were cached (requiring 512 KB for this experiment), the interposed requests would be unnecessary, reducing the overhead for updates to watched blocks to 6–9%.

For some updates that generate an alert, it may be necessary to modify the internal IDD structures to reflect the update. For example, given the rule *Warn me if anything changes in the directory /sbin*, if the file `/sbin/newfile` is created it is perhaps appropriate to both generate an alert and

start watching the new file for subsequent changes. This can either be done by reconstructing the exact change caused by the alert—which may require additional interposed reads—or by reinvoking the full initialization process.

These results show that the overhead involved with determining whether to generate alerts during watched block updates is not unreasonable, especially if it is assumed that such updates occur very infrequently. In the following two subsections we analyze desktop traces from a university laboratory environment to give some insight into the validity of this assumption.

7.5.4.1 Frequency of alert-generating updates

To understand the frequency of overheads beyond the common-case performance, we examined 11 months worth of local file system traces from managed desktop machines in a mid-sized research group. The traces included 820,145,133 file operations, of which 1.8% translated into modifications to the disk. Using these traces, we quantify the frequency of two cases: actual rule violations and non-rule-violating updates, the latter of which are purely updates to shared inode blocks or directory blocks).

We examine the traces for violations of the storage-based IDS rule-set published by Pennington et al. [125]. This rule-set includes the default rules in Tripwire for Red Hat Linux, with additional rules regarding hidden names and append-only audit logs. When applied against our disk image, rules were set on 29,308 files, which resulted in approximately 225,000 blocks being watched. In the traces, 5350 operations (0.0007% of the total) impacted files with rules set on them. All but 10 of these were false positives resulting from regularly-scheduled nightly updates to configuration files such as `/etc/passwd` and regular updates to system binaries. These false positives can be eliminated by making the IDD aware of expected updates—special preparatory commands can be sent by the administrative console in advance of the regular update. This approach has the added benefit that the IDD can provide a helpful confirmation to the administrator (instead of a false positive) indicating that the write completed successfully and that the IDD is aware of the new configuration.

7.5.4.2 Frequency of non-alert-generating updates

The first class of non-rule-violating updates that require ACF execution is shared inode blocks. Our prototype notices any changes to an inode block containing a watched inode, so it must also determine if any such modification impacts an inode being watched. In the case of the ext2 file system, 32 inodes are stored in each inode block. If any inode in a given block is watched, an update to one of the 31 remaining inodes will incur some additional overhead. To quantify this effect, we looked at the number of times an inode was changed which was in the same block as a watched inode. For this analysis, the local file systems of 15 computers were used as examples of which inodes share blocks with watched files. For our traces, 1.9% of I/Os resulted in changes to inode blocks. Of these, 8.1% update inode blocks that are being watched (with a standard deviation

of 2.9% over the 15 machines), for a total of 0.15% of I/Os requiring ACF execution. Most of the inode block overlap resulted from these machines' `/etc/passwd` being updated nightly. This caused its inode to be in close proximity with many short-lived files in `/tmp`. On one machine, which had its own partition for `/tmp`, we found that only 0.013% of modifications caused writes to watched inode blocks. Using the values from Table 7.1, we compute that the extra work would result in a 0.01–0.04% overhead (depending on the IDD cache hit rate).

Similarly, the IDD needs to watch directories between a watched file and the root directory. We looked at the number of namespace changes that the IDD would have to process given our traces. Using the same traces, we found that 0.22% of modifications to the file system result in namespace changes that an ACF would need to process in order to verify that no rule was violated. Based on the measurements in Table 7.1, these ACF invocations would result in a 0.02–0.06% performance impact, depending on IDD cache hit rate.

7.6 Extending real disk products to include IDS capabilities

Workstation disks are extremely cost-sensitive components, making feature extensions a tough proposition. Security features, however, are sufficiently important and marketable today that feature extensions are not impossible. To make for a viable business case, uninterested customers must observe zero cost. The cost of any hardware support needed must be low enough that the profits from the subset of customers utilizing (and paying for) the IDS features must compensate for the marginal hardware costs incurred on all disks produced. A similar situation exists in the network interface card (NIC) industry, where 3Com Corporation embedded sufficient hardware in their standard NICs to allow them to sell firewall-on-NIC extensions to the subset of interested security-sensitive customers [1]; the purchased software is essentially an administrative application that enables the support already embedded in each NIC [2] plus per-seat licenses.

Evaluation of our disk-based IDS prototype suggests that IDS processing and memory requirements are not unreasonable. In the common case of no ACF invocations, even with our untuned code, we observe just a few thousand cycles per disk I/O. Similarly, for a thorough rule-set, the memory required for IDS structures and sufficient cache to avoid disk reads for non-alert ACF executions (e.g., shared inode blocks) is approximately two megabytes. Both are within reasonable bounds for modern disks. They may slightly reduce performance, for example by reducing the amount of disk cache. The overall effect of such changes should be minor in practice, since host caches capture reuse while disk caches help mainly with prefetching. Moreover, neither the memory nor the CPU costs need be incurred by any disk that does not actually initialize and use its IDS functionality.

In addition to the IDS functionality, a disk-based IDS requires the disk to be able to perform the cryptographic functions involved with the secure administrative channel. This requires a key management mechanism and computation support for the cryptography. Again referring to the 3Com NIC example, these costs can be very small. Further, various researchers have proposed the

addition of such functionality to disks to enable secure administration of access control functions [4, 69], and it can also be used to assist secure bootstrapping [10].

7.7 Summary of this chapter

Storage-based intrusion detection is a promising approach, but it would be most effective if embedded in the local storage components of individual workstations. From experiences developing and analyzing a complete disk-based IDS, implemented in a disk emulator, we conclude that such embedding is feasible. The CPU and memory costs are quite small, particularly when marginal hardware costs are considered, and would be near-zero for any disk not using the IDS functionality. The promise of enhanced intrusion detection capabilities in managed computing environments, combined with the low cost of including it, makes disk-based intrusion detection a functionality that should be pursued by disk vendors.

This work represents an example of evaluating system-level interactions with novel storage device functionality, which is one of the types of experiment that is enabled using the technique of timing-accurate storage emulation. A variety of additional evaluation permutations are available using other experimental approaches enabled by this technique. For example, our use of a real disk in the emulator's timing manager could be replaced by a physical device model of a futuristic storage device, in order to acquire an understanding of how the performance impact will change by the time devices with programmable functionality are finally available. Overall, this example of implementing a disk-based IDS using an extended storage interface demonstrates a strength of timing-accurate storage emulation as an evaluation approach.

The following chapter concludes this dissertation by looking to the future of timing-accurate storage emulation.

CHAPTER 8
CONCLUSION

We conclude this dissertation by summarizing the importance of timing-accurate storage emulation, discussing what will be required for timing-accurate storage emulation to achieve widespread use in computer systems analysis, presenting some of the implications of this research, and identifying several promising avenues for continuing work.

8.1 Summary: The importance of timing-accurate storage emulation

Existing techniques are unsatisfactory for evaluating the role of hypothetical storage components in computer systems. These techniques are unable to evaluate true real-system workloads in the context of real system components. They are especially unable to achieve an analysis of real interactions between modified host systems and modified storage device firmware components or novel physical device characteristics. This dissertation describes the technique of timing-accurate storage emulation and its role in mitigating many of the limitations of alternative evaluation techniques.

8.2 Keys to the widespread use of timing-accurate storage emulation

The benefits of timing-accurate storage emulation as an evaluation technique will not come without several costs regarding the development and maintenance of an emulation infrastructure. These costs include those of developing accurate and computationally inexpensive physical device models for interesting hypothetical devices, extending and stabilizing target-mode functionality across a wider range of storage interconnects and emulation operating systems, and creating a broader set of evaluation workloads that are more representative of the systems ultimately to be deployed.

For timing-accurate storage emulation to remain effective, new storage device models need to continue to be created for existing and prospective storage products. Emulation experiments require validated or high-confidence physical device models in order to provide useful experimental results. As discussed before, these can be developed either by individual product manufacturers in the context (hopefully) of a common or standardized emulation environment, or can be developed by academic or product-independent laboratories under governmental or corporate alliance grants. This is not likely to be a problem, since emulation-based experiments use the same physical device models that are used for the other evaluation techniques of storage subsystem simulation and full-system simulation, and historical experience suggests that there is sufficient interest to continue developing models for these environments. As emulated storage devices experiments grow more complex—for example, emulating multi-device components like disk arrays or devices with multilevel caches—an interesting question will arise regarding whether the best architecture involves

building a single instance of an emulator that models all emulated components internally, or using multiple emulated devices connected via the appropriate real bus architecture.

To expand the applicability of emulation-based experimentation, there will also need to be continued development on hardware and software support for target-mode operation across existing interconnects and new interconnects such as Serial Attached SCSI. Historical analysis suggests that there has been interest for target-mode SCSI support in various operating systems for over a decade (and perhaps even longer), and the FreeBSD-based software architecture used herein continues to be actively developed and maintained by the contributors to the FreeBSD project. New hardware support for target-mode operation can be general-purpose, as are the bus adapters described in this dissertation, or can be an extension of the currently-available (non-timing-accurate) storage emulation products that are used either for system-level testing and validation and for building "solid state disks" (large aggregations of high-speed RAM memory resources that are accessed through an ordinary disk-like interface).

Additionally, a much richer set of application-level workloads will need to be developed in order to extract the full usefulness of timing-accurate storage emulation. The lack of diverse and representative workloads for storage evaluation has been and continues to be a problem in the storage systems community, with some organizations having a large set of proprietary workloads but there being few well-planned, well-accepted workloads. We note tongue-in-cheek that authors looking for citation strength can do well by creating and publishing any reasonable workload to fill this void. There has at least been recent interest in aggregating available traces of real-system workloads into a single publicly-available database, the success of which could potentially lead to better characterizations of real-system workloads [88, 89] and ultimately better benchmark and microbenchmark utilities to exercise storage devices and emulated storage devices.

8.3 Implications of this research

There exists a need in computer systems today to make better use of the communications interface between host systems and peripheral components such as locally-attached storage and network devices. This interface is often a demarcation point in computer systems; system designers typically provide for isolated computational hardware and data management on each side of the interface. Many of our results argue the advantages of explicit information sharing across this narrow junction. The packaging and exporting of select characteristics from each side—in effect, sharing additional information across the existing communications paths—provides opportunities to enhance data management, device configuration, and system efficiency. In this way, the complete system becomes greater than the sum of its parts.

Much of the author's research (presented in this dissertation and published elsewhere [125, 142, 146]) takes advantage or could beneficially take advantage of sharing information across the storage interface between operating systems (OSes) and storage devices. The results demonstrate that OSes and storage devices can indeed take more appropriate actions on both sides of the storage

interface by exchanging information. For example, only the OS has detailed information about which applications generate disk requests and the priorities of different requests. On the other hand, only the storage device firmware has exact information about its internal firmware algorithms and current mechanical state. Sharing this information appropriately allows both sides to tune their activities and yields a more functional and efficient storage system.

Additionally, Acharya, Uysal, and Saltz [3], Ganger [58], Riedel [129], Sivathanu et al. [156], Wang, Anderson, and Patterson [171], Wickremesinghe, Chase, and Vitter [173], and others advocate the inclusion of externally-controlled autonomous processing capabilities into standalone storage devices. When considering the deployment and online use of such an "intelligent" device, there will likely be a need to transmit data processing instructions or download executable software into the device, and after execution to retrieve specific results or the execution status from the device; there is currently no standardized method for conveying this information between the two entities. Although specialized interfaces to object-based storage devices provide a potential solution to this problem, the widespread use and universal support of the current block-based interface to storage indicate that such an environment should additionally be considered when implementing intelligent functionality. Regardless, there is a tension between on one hand needing more expressive interfaces to the device to convey this information, and on the other hand needing to keep the existing well-supported and well-understood interfaces intact.

Timing-accurate storage emulation offers the opportunity to investigate these novel uses of storage in computer systems, permitting forays into the space of hypothetical device functionalities without the difficulties of developing and supporting extensively nonstandard or novel interface actions in prototype or production device firmware. This is especially relevant when considering the recent trend toward the development of small-volume, high-capacity, application-specific consumer storage devices for personal and portable computing, especially in the context of audio and video recording and playback, where support for specific features in individual product offerings will play a key role in the advantageous positioning of products in the market.

8.4 Opportunities for future work

This work raises a number of questions whose pursuit may be of interest to researchers or developers that use the technique of timing-accurate storage emulation. In this section we discuss groups of these questions centered around the themes of data management, timing management, and the general use of emulation in computer system evaluations.

8.4.1 Support for large storage working sets

As demonstrated by the evaluations in this dissertation, interesting experimentation can be achieved using only the high-speed resources available to the emulation software. However, One important area for continuing investigation into timing-accurate storage emulation involves pushing the

boundaries of what workloads can be supported by the data manager in a timing-accurate storage emulator. In particular, support for working set sizes larger than the available RAM on the emulation system will require support for the appropriate loading and unloading of data objects between the high-speed and low-speed, resources, as discussed in in Section 3.3.3 (page 37).

Standard techniques for efficient cache management are applicable in the operation of the data manager. In-memory data compression techniques, such as the coalescing of duplicate objects [19, 128] or the keeping of minimal differentials between similar objects [46], may help an emulator make the best use of the high-speed resources, with a limitation that the processing requirements of these techniques (e.g., data decompression and checksum or hash calculations) must be achievable within the relatively short lifespan of an emulated request. A more exhaustive list of such techniques is provided by Kulkarni et al. [106]. An emulator can also use application-specific knowledge in its resource management schemes, including application-directed cache preloading hints [26, 123] and explicit knowledge of the file system structures stored on the emulated device [156]. These techniques may be useful as the data manager attempts to determine when data objects will (or will no longer) be required by the application workload, or attempts to correlate which data objects are likely to be accessed together.

Beyond the usual cache-management techniques, there are opportunities for efficient management of high-speed and low-speed resources that take advantage of specific characteristics of emulation-based experimentation. One such opportunity involves identifying and exploiting any determinism present in the experimental workloads to predict what data will be accessed during future experimentation. Many performance-oriented benchmarks—both those that exercise file systems and those that exercise other computer system components—share the characteristic of repeatable execution from a known initial state resulting in deterministic reproduction of the experimental results. This repeatability is what enables experimental results to be portably compared across system configurations with various different components. The storage traffic created by benchmark workloads may also have a high degree of determinism, both in terms of which blocks are accessed and in which order the accesses occur. When this is true for a particular experiment, an emulator can effectively use a trace of request arrivals from a previous run to correctly manage the loading of the high-speed resources from the low-speed resources during an experiment.

Another opportunity to take advantage of specific characteristics of emulation-based experimentation involves modifying the data that is written by the experimental applications in order to simplify the operation of the data manager. We expect timing-accurate storage emulation to often be applied in laboratory environments[8.1] where it is feasible to modify the experimental applications so long as such modifications do not affect the application behavior. The application can be instructed to write highly compressible data whenever possible, which will enable the emulation software to

[8.1] This is as opposed to being used in a production environment, although recent interest in emulating the functionality of older mainframe hardware using newer computers suggests that there may be applications of timing-accurate storage emulation in filling in for older storage devices used by archaic business-critical applications whose performance is tuned to match that of a particular device.

quickly identify and compress such data during experimentation. As examples of this, file system benchmarks that create files filled with random bytes can be modified to instead create zero-filled files.[8.2] Or, a database storage manager can be instructed to write similar patterns when writing out tables that will not be read again during an experiment.

In general, the key to effective data management for large working sets is to identify which pieces of information are the most helpful for a particular workload, and exploit the appropriate techniques to the maximum gain of the experimenter. Emulation has the advantage that the experimental environment doesn't have to be general-purpose—you can load the dice as much as possible, so to speak—with the additional advantage that failed experiments can generally be re-run taking into account additional knowledge about what failed previously.

8.4.2 Quantifying the real-world representativeness of a device model

As discussed in Section 3.2.4, the real-world representativeness of the timing-accurate emulation of a device model depends in part on the correctness of the model's behavior when driven by the experimental workload. Common problems the authors have encountered in past experimentation using device simulation models include improper device cache modeling and improper handling of large sequential requests (64–128 KB), both of which were only discovered when comparing the results from simulation with the results from experiments with a real disk. Although these simulation models were "validated," further investigation revealed that the validation studies used workloads that were not similar to the workloads used during our experimentation.

As a potential step toward solving such problems, we implore the storage community to develop a standardized approach or methodology for the validation of storage device models, and as part of this to provide an authoritative and portable repository of many storage workloads that are representative of real-world workloads. The centralized availability of a diverse set of input workloads will enable model builders to more easily build more accurate and more widely-applicable storage models. More relevant to timing-accurate storage emulation, this emphasis on standardization could also encourage development of a standardized method for publishing validation results representing the real-world representativeness of the device model, which in turn could be used by emulation software—or any experiment using device models, including those outside the realm of timing-accurate storage emulation—to quantify the overall correctness (or incorrectness) of the results of an emulation-based experiment.

As an alternative approach, the metric determining of the representativeness of a device model could be tied in with a measure of fidelity of the model. As discussed in Section 5.1.1, a device model may support multiple degrees of fidelity, each building on a tradeoff between the accuracy of the model and the resource consumption when executing the model. One possible approach to using fidelity involves the model builder including selectable components (such as scheduling or

[8.2] Writing zero-filled files may trigger undesirable caching inside the host operating system; in this case the application may be modified to write an unusual marker pattern such as `0x8badf00d`.

cache algorithms, or detailed models of mechanical movements) that have been validated to behave correctly and precisely under certain workloads (e.g., small, random read requests) but known to be incorrect under others (large streaming writes). When servicing a workload for which no precise and validated component is available, the simulation model could fall back on default performance approximations (such as table-based lookups) and report this status change to the emulator. The frequency and potential impact of these situations could be tracked by the emulation software and reported post-experimentally as a quantified reduction in the real-world representativeness of the results.

8.4.3 Design choices for timing management

Our choice of the Linux SCSI subsystem mid-to-low-level interface as measurement point MP_1 for the host system (as discussed in Section 5.2.1, page 51) exhibited an intriguing disadvantage whenever the emulated device was not mounted synchronously by the operating system. We discovered that the transmission of write requests across the storage interconnect was often delayed by several milliseconds somewhere below our measurement point (i.e., somewhere along the request critical path, after the `queuecommand()` call). We believe this is intentional behavior, caused by write-coalescing functionality in the OS intended to improve the performance of non-blocking sequential writes by applications.[8.3] Unfortunately, this severely impacts the comparisons of observed times with simulator-computed times and eliminated the usefulness of the error-reduction validation. To accomplish this, further study is required on the nature of the delays between the host and emulator clocks. It may be possible to eliminate the measurement point in the host system during workload evaluation and still be able to validate the experimental results. Our experience is that this error is roughly constant for both storage interconnect adapters, as shown by the results in Section 5.4.2. The ultimate goal of eliminating MP_1 is to enable the pre-experimental calibration and post-experimental validation to be performed using only measurement data from the emulation system.

A related issue involves to what degree synchronization is possible between the clocks on the host and emulator systems. In Equation 5.10, we assumed $E^A_{1\to3} = E^C_{1\to3} = E_{1\to3} \div 2$. If better synchroneity between the clocks is obtained, a nonequal division—such as $E^A_{1\to3} = 0.7 \times E_{1\to3}$ and $E^C_{1\to3} = 0.3 \times E_{1\to3}$—may result in better emulator accuracy during experimentation.

Two additional speculative areas for investigation are introduced by the discussion of the timing-accurate execution of a storage device model in Section 5.1.1 (page 46). One question involves the need for precision of both interrequest and intrarequest timings. There may be situations where it is acceptable for the emulator to not meet a deadline for a particular request, as long as one or more future requests complete early to compensate for the delay. Such graceful recovery would result in an average cumulative error of near zero, but may result in incorrect application behavior due

[8.3] Further, we speculate that similar delays in the host operating system may be a primary reason behind the variable large delays exhibited by the requests in our experiments in Chapter 6, as discussed in Section 6.2.1 (page 98).

to closed-loop timing issues with the host system. Along similar lines, the question remains open concerning whether non-timing-accurate emulation of intrarequest events yields valid overall emulation results, especially when the experimental results re focused on interpreting the performance of a particular external component such as the host system bus adapter.

8.4.4 Interactions with timing-accurate storage emulators

There are several hardware configuration questions that arise when considering the use of a timing-accurate storage emulator. For example, the timing-accurate emulation of a multidevice storage component (such as a disk array) can be accomplished either by using a single emulator containing an internal model of the entire component, or by using multiple emulators that are configured in a representative architecture with identical intraconnections within the component. In the latter case, it remains to be seen whether the overall externally-visible error equals the sum of all the individual emulator errors (caused, for example, by the $E_{1\rightarrow 2}$ communications delays between the emulators and by interdependencies between requests on the individual emulated devices) or simply equals the maximum of the individual emulator errors.

Another hardware configuration question involves the exploitation of extended interfaces to emulated storage devices. As discussed in Section 7.5.1 (page 128), one of the open questions involves accurately modeling the timings of hardware that doesn't match the hardware used by the emulation system. For example, a real device would probably have a specialized embedded processor, whereas the emulation system will likely be based on a general-purpose processor. Two possible solutions to this mismatch include building a tighter interplay between the storage emulator and a real embedded processor to handle the calculations and timings therein, or using a timing-accurate processor emulator in conjunction with the storage emulator—which may lead into investigating the applicability of timing-accurate emulation of other computer system components.

The notion of modifying the data written by an experimental application to simplify the task of data management for a timing-accurate storage emulator is discussed in Section 8.4.1 (page 137). This may be generally considered the notion of *changeable* data, where the contents of the data objects can be changed by either the application or the emulator without affecting the application performance. (The key question involves which data objects can be considered changeable.) This essentially represents a relaxation of the data integrity requirements of a storage device. Extending this thought, another potentially interesting system architecture involves direct communication between the host system and the storage device regarding storage availability requirements for individual data objects. This introduces the concept of what we deem *discardable* data: data for which it is possible (if somewhat inconvenient) for the storage device to lose data without affecting the correctness (but likely affecting the timings) of the workload-generating application. Discardable data are objects that could be reconstructed by the host system. This reconstruction can be based on external information, such as cached files in a web browser, or internal information, such as intermediate object files that are compiled transformations of existing source files. Introducing the

concept of discardable data allows the storage device to practice priority-based scheduling among requests [111] in the highly unconstrained environment of being able to fully ignore writes and reads of at least one class of requests (those reading or writing discardable data). Investigating the use of discardable data in computer systems is straightforward using an expanded interface to the storage device to identify which objects are considered discardable by the application; this ability to explore real systems architectures containing novel storage interactions represents one of the many strengths of timing-accurate storage emulation.

8.5 Availability of the emulation software

The software composing our implementation of the Memulator will be made freely available on the Parallel Data Laboratory web site at the Internet address `http://www.pdl.cmu.edu` following publication of this dissertation. Additionally, the experimental framework and software used to execute the timing experiments for Chapter 5 and Chapter 6 (and an archive of the data generated therein) will be made available at that address.

APPENDIX A
CASE STUDY: EXPERIENCE WITH FULL-SYSTEM SIMULATION

The need for evaluation tools based on timing-accurate storage emulation was borne out of our experiences with two recent projects. These experiences are summarized and discussed as case studies in both this appendix and Appendix B. The purpose of these case studies are twofold: (1) to demonstrate the real-world applicability of results that are obtainable using the techniques of timing-accurate storage emulation, and (2) to demonstrate the limitations of existing storage evaluation techniques that can be overcome through the use of timing-accurate storage emulation.

This appendix explores the system-level applicability of considering a new class of non-volatile storage devices, MEMS-based storage, in their potential role as a replacement or supplement to locally-attached disks. A fuller treatment of the concepts in this section, extending beyond the scope of timing-accurate storage emulation, is available in several conference papers [29, 73, 74, 145, 146, 147] and a dissertation by Steve Schlosser [144].

A.1 Overview of of MEMS-based storage devices

Microelectromechanical systems (MEMS) are very small-scale mechanical structures—on the order of tens to thousands of microns—fabricated on silicon chips using photolithographic processes much like those employed in manufacturing standard semiconductor devices. MEMS structures can be made to slide, bend, or deflect in response to an actuator's electrostatic or electromagnetic force or external forces. MEMS machines have interesting strengths and limitations compared to standard mechanical systems. For example, large-aspect-ratio cantilever designs that would fail under load when built at the macroscopic scale can be built reliably on the microscopic scale. As a counterexample, it is difficult to build durable microbearings for rotating components—prototypes of micromachined gear trains have locked up from friction within several thousand revolutions. Because of this limitation it is difficult to replicate disk-based storage designs on the microscopic scale. Alternative designs, such as rectangular spring-suspended masses (*media sleds*) that translate two-dimensionally (instead of rotating about an axis), circumvent this frictional barrier and are proving to be mechanically robust.

One class of MEMS-based storage device under investigation employs an array of thousands of cantilevered magnetic read/write heads (*probe tips*, shown in Figure A.1), each accessing a dense substrate of magnetic material in much the same way disk heads access magnetic platters [23, 28]. This design offers notable advantages over disk-based storage along several axes, including access time, device size and mass, energy consumption, cost, failure modes, and sensitivity to shock. Multiple probe tips can concurrently access the media to achieve one of several forms of parallelism: all

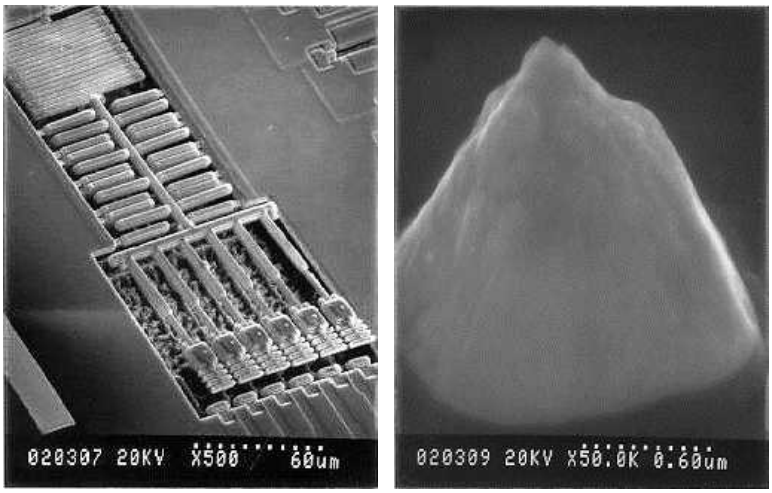


Figure A.1: Prototype positioning system and probe tip for MEMS-based storage. *Because the recording material is not perfectly flat, the positioning system must be able to actively adjust the height of the probe tips. The tips could use one of several recording schemes, from simple "typewriting" with permanent magnets, to more complex magnetoresistive sensing techniques found in normal disk drives.*

tips can be used to access data (to increase throughput); some tips can be used for error detection and correction (to enhance reliability); or completely independent accesses can proceed in parallel. In addition, the MEMS fabrication process can be integrated with standard CMOS processes [51], opening the door to combine processing and non-volatile storage for large-scale manufacturing of system-on-a-chip architectures.

MEMS microstructures can be used to build storage devices in a variety of ways—design decisions affect the manufacturability, robustness, cost, capacity, access speed and latency of these devices. Figure A.2 depicts one proposed MEMS-based storage design. In this "fixed media" model, miniature cantilevered L-shaped beams suspend a probe tip over a fixed magnetic substrate. Voltages applied to deflectors generate electrostatic forces in the X and Y directions, rapidly moving the tip to different bit positions. Standard magnetic recording techniques are used to read or write the bits, with the same unlimited number of read and write cycles as found in disk drives. The nearly-massless cantilevered beam enables very quick positioning times (on the order of tens to hundreds of microseconds) but the space efficiency is poor—only about 1% of the potential media area can be used for storage. In comparison, conventional disk drives use about 50% of their platter area for data storage. This design is useful for visualizing MEMS-based storage, but its expected capacity of only tens to hundreds of megabytes per device limits its practicality in comparison to Flash RAM, battery-backed RAM, and other non-volatile primary storage components.

Researchers at Carnegie Mellon are investigating a more media efficient device design, which is shown in Figure A.3. In this "moving media" model, a rectangular media sled is suspended by springs above an array of several thousand fixed probe tips. A device's footprint is about 14×14 mm,

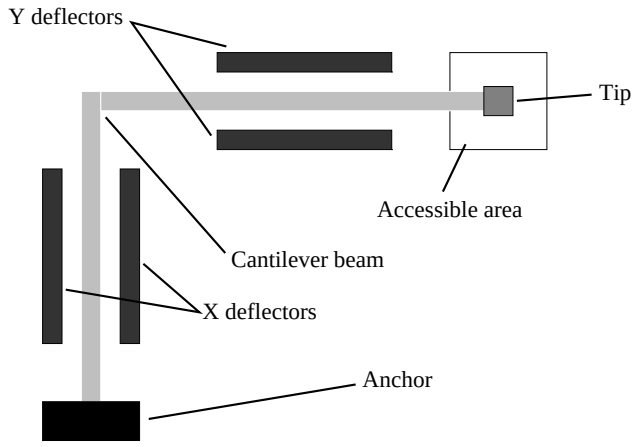


Figure A.2: A cantilevered-beam probe tip in the "fixed media" model for MEMS-based storage. *The X- and Y-deflectors are capable of quickly positioning the tip anywhere in the small accessible area. The overall capacity of this model is limited to tens or perhaps hundreds of megabytes because only 1% of the media area is accessible by the tip.*

with a usable area on the media sled of about 8×8 mm. Up to 10,000 tips can be fabricated over this 8×8 mm area. Assuming a bit cell of 0.0025 μm^2 (50 nm per side) and encoding/ECC overheads of 2 bits per byte, a device's data storage capacity is about 4 GB [28]. A more aggressive goal of 0.0009 μm^2 (30 nm per side) yields capacities of 11 GB or greater. While this device design improves space efficiency to 30–50%, the greater sled mass increases positioning times relative to the fixed media design above—a necessary tradeoff to achieve disk-like capacities. Carley et al. [28] and Griffin et al. [73] provide a more thorough description of the characteristics of this design.

A.2 Device data layout and access characteristics

The sled's magnetic media is organized into rectangular regions as shown in Figure A.4. Each region stores M×N bits (e.g., 2000×2000). There is a one-to-one mapping between regions and tips; each tip accesses its exclusive region of the media. Bits within a region are grouped into vertical 90-bit columns called *tip sectors*; each tip sector contains 10 bits of sled positioning information and 80 encoded data bits providing 8 data bytes. The 8-byte tip sector is the smallest accessible unit of data in MEMS-based storage. Groups of 64 tip sectors from separate regions may be combined into 512-byte *logical sectors*, analogous to logical blocks in SCSI disks. This striping is both possible and practical because, unlike most conventional disks, large numbers (200–2000) of probe tips can simultaneously access the media. Striping logical blocks across tip sectors in multiple regions reduces access time and increases bandwidth, reliability, and fault tolerance.

To access data, electrostatic actuators (capacitive comb fingers) pull the sled to a certain *x*,*y* offset—positioning the tips above an exact location on the media by moving the media—then drag the sled such that each active tip reads or writes an entire tip sector (*i.e.*, such that groups of tips

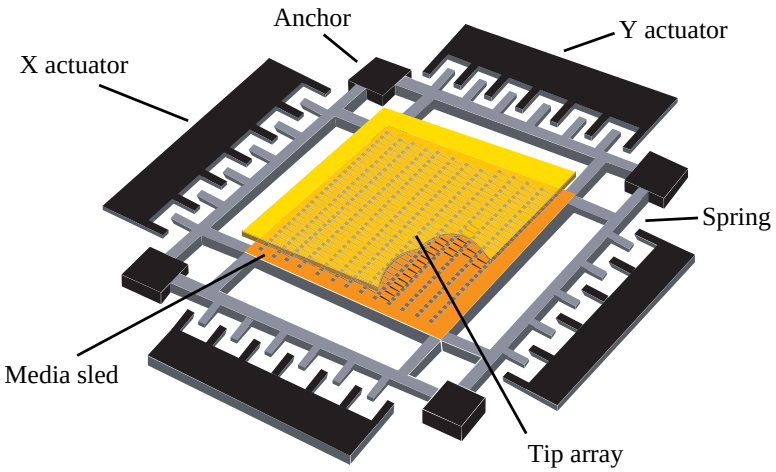


Figure A.3: The "moving media" model for MEMS-based storage. *The media sled is attached below the fixed tips. The sled can move along the X and Y axes, allowing the fixed tips to address 30–50% of the total media area.*

access whole logical sectors). As in the earlier design, the probe tips read or write data using standard magnetic recording techniques.

Positioning the sled for read or write involves several mechanical and electrical actions. To seek to a desired sector, the appropriate probe tips must be activated, the sled must be positioned so the tips are above the first bit of the pre-sector servo information, and the sled must be moving in the correct direction at the correct velocity for access. Whenever the sled moves in X, an extra constant *settling time* must be taken into account—the rapid acceleration and deceleration of the sled causes the spring-sled system to momentarily oscillate in X before damping to zero X motion. (The sled also oscillates in Y; however, the magnetic sensing logic is expected to compensate for this motion.) In addition, the springs apply a restoring force toward the "sled-at-rest" position, increasing or decreasing the effective sled actuating force by as much as 75%.

Media access requires that the sled move at constant velocity in the Y direction. This *access velocity* is a design parameter and is determined by the maximum per-tip read and write rates, the bit width, and the maximum sled acceleration. Large transfers could span multiple columns of bits, requiring the sled to perform a *turnaround* (reversing direction such that the sled ends up in the same position at reverse velocity) and switch the set of active tips. The turnaround time is expected to dominate any additional activity, such as the time to switch which tips are active.

A.3 Contrasting MEMS-based storage devices with disks

Although MEMS-based storage devices involve some radically different technologies from disks, they share enough fundamental similarity for a disk-like model to be a sensible starting point. This section compares MEMS-based storage devices and disks from this standpoint, and the rest of the dissertation shows that little is lost by taking this view.

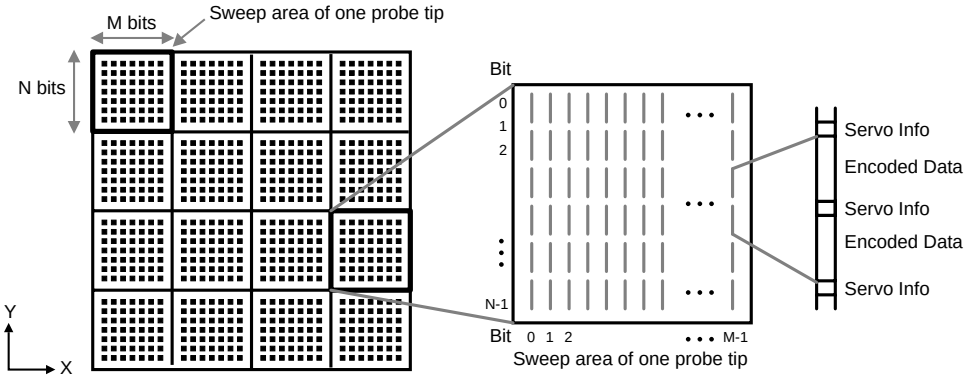


Figure A.4: Physical data organization for MEMS-based storage. *The illustration depicts a small portion of the magnetic media sled. Each rectangle outlines the region accessible by a single probe tip, with a total of 16 regions shown. (A full device contains thousands of tips and regions.) Each region stores $M \times N$ bits, organized into vertical "tip sectors" containing encoded data and ECC bits. These tip sectors are demarcated by "servo information" strings that identify the sector and track information encoded on a disk. This servo information is expected to require about 11% of the device capacity. To read or write data, the media passes under the active tip(s) in the $\pm Y$ direction while the tips access the media.*

Like disks, MEMS-based storage devices stream data at a high rate and suffer a substantial distance-dependent positioning time delay before each nonsequential access. In fact, although MEMS-based storage devices are much faster, they have ratios of request throughput to data bandwidth similar to those of disks from the early 1990s. Some values of the ratio, γ, of request service rate (requests/s) to streaming bandwidth (MB/s) for some recent disks include $\gamma = 26$ (1989) for the CDC Wren-IV [122], $\gamma = 17$ (1993) [78], and $\gamma = 5.2$ (1999) for the Quantum Atlas 10K [127]. γ for disks continue to drop over time as bandwidth improves at a greater rate than mechanical positioning times. In comparison, the MEMS-based storage device described below yields $\gamma = 25$ (1111 requests/s $\div$ 44.8 MB/s), comparable to disks within the last decade. Also, although many probe tips access the media in parallel, they are all limited to accessing the same relative *x*,*y* offset within a region at any given point in time—recall that the media sled moves freely while the probe tips remain relatively fixed. Thus, the probe tip parallelism provides greater data rates but not concurrent, independent accesses. There are alternative physical device designs that would support greater access concurrency and lower positioning times, but at substantial cost in capacity [73].

The remainder of this subsection enumerates a number of relevant similarities and differences between MEMS-based storage devices and conventional disk drives.

Mechanical positioning. Both disks and MEMS-based storage devices have two main components of positioning time for each request: seek and rotation for disks, X and Y dimension seeks for MEMS-based storage devices. The major difference is that the disk components are independent (i.e., desired sectors rotate past the read/write head periodically, independent of when seeks complete), whereas the two components are handled in parallel for MEMS-based storage devices. As

a result, total positioning time for MEMS-based storage devices equals the greater of the X and Y seek times, making the lesser time irrelevant. This overlap most strongly affects request scheduling.

Settling time. For both disks and MEMS-based storage devices, it is necessary for read/write heads to settle over the desired track after a seek. Settling time for disks is a relatively small component of most seek times (0.5 ms of 1–15 ms seeks). However, settling time for MEMS-based storage devices is expected to be a relatively substantial component of seek time (0.2 ms of 0.2–0.8 ms seeks). Because the settling time is generally constant, this has the effect of making seek times more constant, which in turn could reduce (but not eliminate) the benefit of both request scheduling and data placement.

Logical-to-physical mappings. As with disks, the lowest-level mapping of logical block numbers (LBNs) to physical locations will be straightforward and optimized for sequential access; this will be best for legacy systems that use these new devices as disk replacements. Such a sequentially optimized mapping scheme fits disk terminology and has some similar characteristics. Nonetheless, the physical differences will make data placement decisions (mapping of file or database blocks to LBNs) an interesting topic.

Seek time vs. seek distance. For disks, seek times are relatively constant functions of the seek distance, independent of the start cylinder and direction of seek. Because of the spring restoring forces, this is not true of MEMS-based storage devices. Short seeks near the edges take longer than they do near the center. Also, turnarounds near the edges take either less time or more, depending on the direction of sled motion. As a result, seek-reducing request scheduling algorithms [179] may not achieve their best performance if they look only at distances between LBNs on MEMS-based storage devices, as is commonly done for disks.

Recording density. Some MEMS-based storage devices use the same basic magnetic recording technologies as disks [28]. Thus, the same types of fabrication and grown media defects can be expected. However, because of the much higher bit densities of MEMS-based storage devices, each such media defect will affect a much larger number of bits.

Numbers of mechanical components. MEMS-based storage devices have many more distinct mechanical parts than disks. Although their very small movements make them more robust than the large disk mechanics, the sheer number of parts makes it much more likely that some number of them will break. In fact, manufacturing yields may dictate that the devices operate with some number of broken mechanical components.

Concurrent read/write heads. Because it is difficult and expensive for drive manufacturers to enable parallel activity, most modern disk drives use only one read/write head at a time for data access. Even drives that do support parallel activity are limited to only 2–20 heads. On the other hand, MEMS-based storage devices (with their per-tip actuation and control components) could theoretically use all of their probe tips concurrently. Even after power and heat considerations, hundreds or thousands of concurrently active probe tips is a realistic expectation. This parallelism increases media bandwidth and offers opportunities for improved reliability. Further, flexibility

in the choice of which tips are used to access data allows for novel data access schemes, such as efficient access to two-dimensional data structures.

Control over mechanical movements. Unlike disks, which rotate at a constant velocity independent of ongoing accesses, the mechanical movements of MEMS-based storage devices can be explicitly controlled. As a result, access patterns that suffer significantly from independent rotation can be better served. The best example of this is repeated access to the same block, as often occurs for synchronous meta-data updates or read-modify-write sequences.

Startup activities. Like disks, MEMS-based storage devices will require some time to ready themselves for media accesses when powered up. However, because of the size of their mechanical structures and their lack of rotation, the time and power required for startup will be much less than for disks.

Drive-side management. As with disks, management functionality will be split between host operating systems and device firmware. Over the years, increasing amounts of functionality have shifted into disk firmware, enabling a variety of portability, reliability, mobility, performance, and scalability enhancements. Similar trends are likely with MEMS-based storage devices, whose silicon implementations offer the possibility of direct integration of storage with computational logic.

Speed-matching buffers. As with disks, MEMS-based storage devices access the media as the sled moves past the probe tips at a fixed rate. Since this rate rarely matches that of the external interface, speed-matching buffers are important. Further, because sequential request streams are important aspects of many real systems, these speed-matching buffers will play an important role in prefetching and then caching of sequential LBNs. Also, most block reuse will be captured by larger host memory caches instead of in the device cache.

Sectors per track. Disk media is organized as a series of concentric circles, with outer circles having larger circumferences than inner circles. This fact led disk manufacturers to use banded (zoned) recording in place of a constant bits-per-track scheme in order to increase storage density and bandwidth. For example, banded recording results in a 3:2 ratio between the number of sectors on the outermost (334 sectors) and innermost (229 sectors) tracks on the Quantum Atlas 10K drive [56]. Because MEMS-based storage devices organize their media in fixed-size columns instead, there is no length difference between tracks and banded recording is not relevant. Therefore, block layout techniques that try to exploit banded recording will not provide benefit for these devices. On the other hand, for block layouts that try to consider track boundaries and block offsets within tracks, this uniformity (which was common in disks 10 or more years ago) will simplify or enable correct implementations.

A.4 System-oriented evaluation using full-system simulation

This section presents the results from real-world benchmarks, measured on systems with simulated MEMS-based storage devices in two different configurations: first, as a simple replacement for disks; and second, as a non-volatile disk cache.

	G1	G2	G3
bit width (nm)	50	40	30
sled acceleration (g)	70	82	105
access speed (kbit/s)	400	700	1000
X settling time (ms)	0.431	0.215	0.144
total tips	6400	6400	6400
active tips	640	1280	3200
max throughput (MB/s)	25.6	89.6	320
number of sleds	1	1	1
per-sled capacity (GB)	2.56	4.00	7.11
bidirectional access	no	yes	yes

Table A.1: Parameters describing the three generations of MEMS-based storage device models used for experimentation. *G1 represents the first-generation model, G2 the second-generation model, and G3 the third-generation model, as discussed in Section A.4.1.*

A.4.1 Description of the devices used for experimental comparison

Given the wide range of parameters, exploring the entire MEMS-based storage design space is not feasible. Instead, three models of MEMS-based storage are used, based on anticipated technology advances over the first three generations. The parameters describing the three generational models are presented in Table A.1.

The *first-generation (G1) model* represents a conservative initial MEMS-based storage device, which could be fabricated within the next three years [28]. The sled has a full range of motion of 100 μm along the X and Y axes, and the actuators accelerate the sled at 70g. To access data, the device uses a relatively primitive recording scheme, leading to a per-tip data rate of 400 kbit/s. This design only supports unidirectional accesses, where reads and writes only occur when the sled moves in the positive Y direction.

G1's media, tip resolution, and sled positioning system provide a square bit cell of 50 nm such that each tip addresses a 2000×2000 array of bits. The sled footprint is 0.64 cm^2 allowing 6400 tips for each sled. This yields a raw capacity of 2.56 GB per sled. However, media error management requires a 10-bit-per-byte encoding. Also, sled tracking and synchronization information requires 10 tracking bits for every 80 data bits. During media access, the sled is restricted to the fixed access velocity. However, the sled speed is not limited during seeks.

The *second-generation (G2) model* represents several fundamental improvements over G1. First, media access occurs in both the $+$Y and $-$Y directions. Second, per-tip data rate increases to 700 Kbit/s based on trends in probe tip technology. A decrease in the sled mass and an increase in the actuator voltage leads to an increase in sled acceleration to 82g. Also, improvement in the servo system reduces the settling time for each X seek. Decreases in per-tip power utilization can lead to a larger number of tips that can be active simultaneously, vastly improving the maximum throughput. Finally, media material improvements increase G2's bit density by 20%.

	Quantum Atlas 10K	Extrapolated SuperDisk
Rotational speed	10,025 rev/min	20,000 rev/min
Maximum bandwidth	25 MB/s	170 MB/s
Data surfaces	6	12
Average rotational latency	2.21 ms	1.36 ms
Average seek (read)	5.7 ms	3.12 ms
Average seek (write)	6.19 ms	3.58 ms
Maximum seek (read)	10.83 ms	8.50 ms
Maximum seek (write)	11.32 ms	8.96 ms

Table A.2: Performance characteristics of the Quantum Atlas 10K disk drive and the extrapolated SuperDisk model.

The *third-generation (G3) model* approaches the high-end of many MEMS-based storage parameters and characteristics. Here the bit density scales down to 30 nm per bit, and a decrease in the sled mass leads to higher sled acceleration. In this case a change in the suspension and sled design leads to a higher resonant frequency, resulting in a shorter X settling time. Throughput is increased, largely because of the addition of more active tips.

The *reference disk* used in our experiments is the Quantum Atlas 10K [127]. The existence of a validated DiskSim module [141] for the Atlas 10K enabled a comparison of a modern disk's performance to MEMS-based storage device performance.

The *extrapolated "SuperDisk" model* was created to compare MEMS-based storage to an aggressive disk drive projection to the year 2005. Extrapolating on the current performance trends in disk drive technology, the SuperDisk achieves streaming bandwidth of up to 125 MB/s. Its seek time drops to a 3 ms average and it rotates at 20,000 RPM. The Atlas 10K and SuperDisk parameters are compared in Table A.2.

Using the model described by Griffin et al. [73] and the device parameters in Table A.1, we created simulation models for each MEMS-based storage device and integrated those models into DiskSim, a freely-available disk simulator that accurately models disk drives [64], including the Atlas 10K. DiskSim was integrated with the SimOS machine simulator [132]. SimOS was configured to model a 1 GHz Alpha 21164-based system with 128 MB of RAM running Digital UNIX version 4. The OS runs atop the virtual machine, using special device drivers to interact with simulated I/O devices. Finally, a model of IBM's low-power disk drive [38] was used to compare against our MEMS-based storage power models. These power models were driven using timing-accurate traces of SCSI block requests gathered from Linux's SCSI device driver.

A.4.2 MEMS-based storage devices as replacements for disks

The first two applications, the Andrew Benchmark Suite [87] and PostMark [96] were designed for file system and I/O performance analysis. The Andrew Benchmark consists of a set of file and directory operations followed by a long compile. The PostMark benchmark performs many small

	Andrew		Postmark		Gnuld		TPC-D #4		TPC-D #6	
	CPU	I/O	CPU	I/O	CPU	I/O	CPU	I/O	CPU	I/O
Atlas 10k	2.8	3.9	9.8	730.4	0.8	25.1	2.7	27.7	8.9	22.3
SuperDisk	2.8	1.7	10.0	397.0	0.7	8.8	2.7	3.3	8.8	0.3
G1 MEMS	2.8	1.8	10.3	257.4	0.8	11.3	2.7	14.8	8.9	5.5
G2 MEMS	2.8	1.0	10.9	171.0	0.8	4.6	2.7	5.2	8.9	0.2
G3 MEMS	2.8	0.7	11.0	170.9	0.8	3.6	2.7	4.2	8.8	0.3

Table A.3: Comparison of five applications on disks and MEMS-based storage devices. *All numbers are in seconds.*

file operations (*e.g.,* create, delete, read, write) and was designed to be representative of the file system workloads seen in e-mail, news, and electronic commerce environments. Table A.3 shows that MEMS-based storage devices can significantly reduce the I/O time for these workloads. Both Andrew and PostMark show an improvement in I/O service time between 4X and 6X, with an overall application performance improvement between 2X and 4X.

The GNU Linker benchmark, Gnuld, is a test in which a large set of object files are linked using the GNU linker. All of the MEMS-based storage devices improve performance over the Atlas10k, with the G3 device decreasing I/O time by 7X. However, SuperDisk's higher bandwidth greatly enhances its performance over the G1 device.

The TPC-D [165] benchmarks also see a large reduction in I/O time from the MEMS-based storage devices. The higher bandwidth of the SuperDisk, however, greatly enhances its performance for the TPC-D queries. In both cases, the SuperDisk out-performs the G1 MEMS device. The performance of the MEMS-based storage devices is also hampered by very high disk cache hit rates for the TPC-D queries, which are between 83% and 90%, respectively, for the disks. Our MEMS-based storage device does not include a prefetching cache, and so cannot benefit from the high sequentiality and data reuse of these benchmarks. However, even without a RAM cache, the MEMS-based storage devices outperform the baseline disk by a wide margin.

A.4.3 MEMS-based storage devices as caches for disks

MEMS-based storage can also be used as an augmentation of the existing storage hierarchy. For example, with their low entry cost, MEMS-based storage devices could be incorporated into future disk drives as very large (1–10 GB) non-volatile caches. The superior performance of MEMS-based storage devices would allow the cache to absorb latency-critical synchronous writes to meta-data and cache small files to improve small read performance. For example, Baker et al. show that using fast non-volatile storage to absorb synchronous disk writes both at a client and at a file server increases performance from 20% to 90% [14].

To explore MEMS-based storage as a non-volatile cache for disk, DiskSim was augmented to allow a MEMS-based storage device to serve as a cache for a disk. The cache was 2.5 GB, the disk was 9.2 GB, and the workload was the 1-day cello trace from [135]. This trace actually includes

eight separate devices so the experiments use a cache per disk. The results show that the average I/O response time is 14.66 ms for an Atlas10K disk drive without any MEMS cache vs. 4.03 ms for a disk with a G2 type MEMS-cache (and 2.76 ms for a single large G2 MEMS device that replaced the disk). Since most of the read requests are serviced from the client-side DRAM cache, the 3.5X performance improvement, over just a disk drive, is achieved mainly by quickly servicing writes. However, unlike DRAM-based write caching (which absorbs writes but risks losing data), the MEMS cache is non-volatile, providing the same data integrity guarantees as disk drives. An alternate experiment in which all eight devices in the cello trace were re-mapped to a larger version of the Atlas10K disk with a single MEMS cache only suffered a slight increase in average access time to 4.66 ms. This longer service time stems from an increase in queueing since the large single device is doing the work of eight. It shows, however, that caching absorbs enough of the device's activity to provide a good performance boost.

Instead of using the MEMS-based storage device as a cache, it is also possible to expose the device to the OS so that file systems can allocate specific data onto it. Depending on their access patterns and performance needs, file systems could place small structures (*e.g.,* file system meta-data) on MEMS-based storage, while using the disk for streamed or infrequently-accessed data. This could be done on individual disks or within RAID arrays, creating the potential for AutoRAID-like systems [175]. Further, because RAID arrays are less cost-sensitive than individual disks, arrays of MEMS-based storage devices could be incorporated more cost-effectively into RAID arrays, providing significant performance improvements for RAID's costly write operations.

A.4.4 Power utilization comparison

The physical characteristics of MEMS-based storage devices may make them less power hungry than even low-power disk drives [92, 93]. This power advantage comes from several sources: lower overall power requirements for moving the media and operating the read/write tips, and faster transitions between active and standby modes.

While the media sled in a MEMS-based storage device does move continuously in the X and Y directions during data access, the sled has much less mass than a disk platter and therefore takes far less power to keep in motion. Specifically, it takes less than 100 mW to continuously move a MEMS sled, while it takes over 600 mW to continuously spin a disk drive.

Another power savings comes from the electronics of MEMS-based storage devices. In disk drives, the electronics span multiple chips and great distance from the magnetic head at the end of the arm to the drive interface. Therefore, high-speed signals must cross several chip boundaries, increasing power dissipation. Further, disks' large physical platters, heads, arms and actuators require sophisticated, power-hungry signal processing algorithms to compensate for imperfect manufacturing, thermal changes, environmental changes, and general wear. Current low-power drives consume almost 1.5 W [92, 93] in drive electronics, much of it spent on accurately positioning the recording head. Of course, not all drive electronics must be active during short idle periods; some electronics,

such as the servo control, can be powered down. This technique reduces total drive power by up to 60%, adding a small additional time penalty to return to active mode (from 40–400 ms).

Drive power can also be saved by turning off the spindle motor during long idle periods. Numerous studies have demonstrated the power savings of this standby mode [48, 86, 108, 110], and current low-power drives do incorporate this feature. MEMS-based storage can also employ a standby mode, stopping sled movement during periods of inactivity. Further, the sled's low mass allows MEMS to quickly switch between active and standby mode (0.5 ms), where a low-power drive requires up to 2 seconds to spin up and return to active mode. This long delay significantly increases access time for the first request after an idle period. Therefore, drive power-management algorithms usually wait at least 10 seconds before going into standby mode. During this 10 second delay, and during the 2 second spin-up time, considerable power is wasted. In contrast, MEMS-based devices can transition from standby-to-active in 0.5 ms, allowing these devices to be much more aggressive in using standby mode.

MEMS-based storage also has the ability to adjust its power consumption during data accesses by reading or writing at a smaller granularity than standard 512 byte blocks. Since most power is dissipated by the probe tips, and not by positioning or moving the media sled, reading or writing only the necessary data could save considerable power. The device only needs to activate as many tips as are necessary to satisfy a request, which could result in a substantial power savings. In contrast, the power required to move a disk drive's arm and spindle, and to servo control the head over the appropriate sector is much greater than the power necessary to actually read or write the 512 byte sector.

To understand how much power a MEMS-based storage device could save over a low-power drive, we simulated both and measured their power consumption across six workloads. The disk drive power model is based on IBM's low-power Travelstar disk and power management techniques described in [92, 93]. The device has 5 power modes: (1) active mode (data is being accessed) consumes 2.5 W for reads and 2.7 W for writes; (2) performance idle (some electronics are powered down) consumes 2.0 W; (3) fast idle (head is parked and servo control is powered down) consumes 1.3 W; (4) low-power idle (heads are unloaded from the disk) consumes 0.85 W; (5) standby (spindle motor is stopped) consumes 0.2 W. According to the disk specification, the maximum time spent in the intermediate modes is: 1 second for performance idle, 3 seconds for fast idle, and 8 seconds for low-power idle [38].

For the MEMS-based storage device, power for a benchmark is computed during simulation by using the physical parameters described by Carley et al. [28]; each probe tip and its signal processing electronics consume 1 mW. To minimize packaging costs, we set our power budget to about 1 W. This limits the MEMS-based storage device to no more than about 1,000 simultaneously active probe tips. Further, given the sled design, the power consumed to keep the sled in motion is 0.1 W. Therefore, the maximum power for this MEMS-based storage device is 1.1 W. Standby power consumption is estimated to be 0.05 W.

	Andrew		Gnuld		Postmark		TPC-D #4		TPC-D #6		Netscape	
	D	M	D	M	D	M	D	M	D	M	D	M
active	19.5	0.7	84.6	3.6	1930.6	42.0	115.6	8.5	59.0	8.4	321.2	1.4
perfIdle	13.3	0.3	39.8	0.0	1181.1	7.7	45.4	0.1	43.6	0.3	1924.1	0.01
goToActive	0.0		0.0		0.0		0.0		0.0		513.5	
fastIdle	0.0		0.0		0.0		0.0		0.0		1799.9	
lowPowerIdle	0.0		0.0		0.0		0.0		0.0		1000.5	
spinup	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	228.8	20.0
standby	0.0	0.2	0.0	0.0	0.0	8.0	0.0	1.1	0.0	1.9	308.9	327.9
Total (Joules)	32.8	1.2	124.4	3.6	3111.7	57.7	161.0	9.7	102.6	10.6	6096.9	349.3

Table A.4: Comparison of energy required to execute six different workloads using disks and MEMS-based storage devices. *The "D" category represents the disk, whereas the "M" category represents the MEMS-based storage device. All numbers are given in Joules.*

Table A.4 shows that the total energy consumed for the MEMS-based storage device is between approximately 10X and 50X lower, depending on the application. The five workloads already discussed are highly active and so most of the savings comes directly from lower energy consumption during data accesses (active mode). To test a more interactive workload, we traced the disk accesses generated by a user browsing with Netscape on a Linux workstation for ten minutes. In this case, much of the power savings comes from MEMS-based storage's ability to aggressively use its low-power standby mode. In contrast, the disk drive spends 90% of its power transitioning between active and standby modes.

A.5 The need for timing-accurate storage emulation

The work described herein analyzes the implications and benefits of introducing hypothetical MEMS-based storage devices into two levels of the computer system memory hierarchy. When used as replacements for disks, standalone MEMS-based storage devices reduce I/O stall times by 4–74X and improve overall application run times by 1.9–4.4X. When used as on-board caches for disks, MEMS-based storage improves I/O response time by up to 3.5X. When used as disk replacements in low-power contexts, the energy consumption of MEMS-based storage is 10–54X less than that of state-of-the-art low-power disk drives. The combination of the high-level physical characteristics of MEMS-based storage (small footprints, high shock tolerance) and the ability to directly integrate MEMS-based storage with processing suggests their applicability in currently popular applications such as portable gigabit storage systems and ubiquitous active storage nodes.

The results in this section were obtained using the techniques of full-system simulation and storage subsystem simulation. The latter technique was necessary due to the experiment-time inability to integrate user interactivity into the experiments, as would be required to recreate the environment used to gather the Cello and Netscape traces. This duality demonstrates two of the limitations of the technique of full-system simulation: first, the difficulty of executing experiments that rely on time-critical external (to the full system) interactions, and second, the inability to use hardware of software platforms for which validated simulation models are not available. Even the simulation

parameters used for these experiments—such as the 1 GHz processor—lay outside the validated parameters for SimOS and likely degrade the realism of the experimental results. The application-oriented experimentation and results described herein would be equally obtainable using timing-accurate storage emulation, with the added advantage that future experimentation would not share these limitations.

APPENDIX B
CASE STUDY: EXPERIENCE WITH PRODUCTION-DEVICE EXPERIMENTATION

As discussed in Appendix A, the purpose of this case study is twofold: to demonstrate the real-world applicability of results that are obtainable using the techniques of timing-accurate storage emulation, and to demonstrate the limitations of existing storage evaluation techniques that can be overcome through the use of timing-accurate storage emulation.

This appendix explores an opportunity to improve the efficiency of operating system access to locally-attached disks, by modifying the operating system's data allocation and access to better match the performance characteristics of real disks. A fuller treatment of the concepts in this section, extending beyond the scope of timing-accurate storage emulation, is available in several conference papers [140, 141, 142, 143] and a dissertation by Jiri Schindler [139].

B.1 The diminishing returns of creating ever-larger disk requests

In determining what data to read and write when, system software attempts to maximize overall performance in the face of two competing pressures. On the one hand, the underlying disk technology pushes for larger request sizes in order to maximize disk efficiency. Specifically, time-consuming mechanical delays can be amortized by transferring large amounts of data between each repositioning of the disk head. For example, Point A of Figure B.1 shows that reading or writing 1 MB at a time results in a 75% disk efficiency for ordinary accesses. On the other hand, resource limitations and imperfect information about future accesses impose costs on the use of very large requests.

File systems and databases attempt to mitigate the ever-present disk performance problem by aggressively clustering on-disk data and by issuing fewer, larger disk requests. This is usually done with only a vague understanding of disk characteristics, focusing on the notion that bigger requests are better because they amortize per-request positioning delays over larger data transfers. Although this notion is generally correct, there are performance and complexity costs associated with making requests larger and larger. For video servers, ever-larger requests increase both buffer space requirements and stream initiation latency [31, 32, 97]. Log-structured file systems (LFS) incur higher cleaning overheads as segment size increases [30, 114, 134]. Even for general file system operation, allocation of very large sequential regions competes with space management robustness [116], and very large accesses may put deep prefetching ahead of foreground requests. Also, large requests can be used for small files by grouping their contents [55, 60, 66, 134], but larger requests require grouping more files with weaker inter-relationships. These examples all indicate that achieving higher disk efficiency with smaller request sizes would be valuable.

System software designers would like to be able to always use large disk requests in order to maximize disk efficiency. Unfortunately, resource limitations and imperfect information about

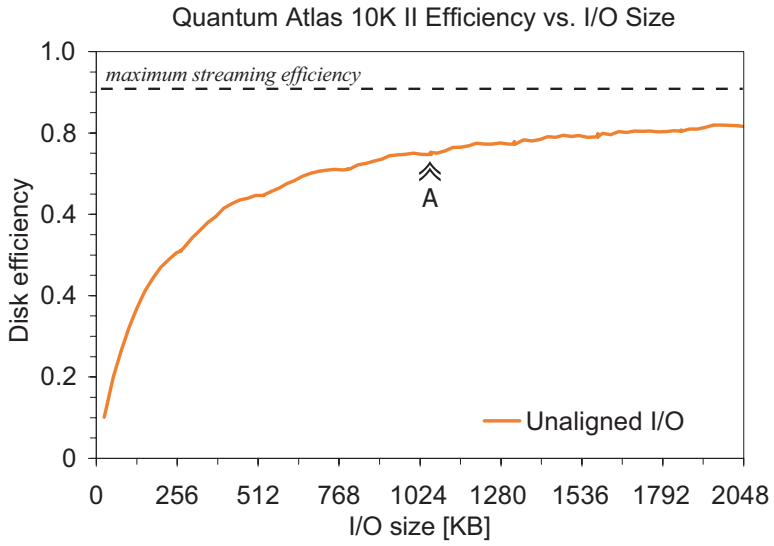


Figure B.1: Increased disk efficiency resulting from larger request sizes. *This graph plots disk efficiency as a function of I/O size. We define "disk efficiency" as the fraction of total access time, including seek and rotational latency, spent moving data to or from the media. The maximum streaming efficiency is less than 1.0, because no data is transferred when switching from one track to the next. Point A shows that reading or writing 1 MB requests results in a 75% disk efficiency for ordinary disk accesses. The "Unaligned I/O" curve shows disk efficiency for random, constant-sized reads within a Quantum Atlas 10K II's first zone (264 KB per track).*

future accesses make this difficult in practice. Four system-level factors oppose the use of ever-larger requests: responsiveness, limited buffer space, irregular access patterns, and storage space management.

Responsiveness. Although larger requests increase disk efficiency, they do so at the expense of higher latency. This trade-off between efficiency and responsiveness is a recurring theme in computer systems, and it is particularly steep for disk systems. The latency increase can manifest itself in several ways. At the local level, the non-preemptive nature of disk requests combined with the long access times of large requests (35–50 ms for 1 MB requests) can result in substantial I/O wait times for small, synchronous requests. This problem has been noted for both FFS and LFS [30, 150]. At the global level, grouping substantial quantities of data into large disk writes usually requires heavy use of write-back caching. Although application performance is usually decoupled from the eventual write-back, application changes are not persistent until the disk writes complete. Making matters worse, the amount of data that must be delayed and buffered to achieve large enough writes continues to grow. As another example, many video servers fetch video segments in carefully-scheduled rounds of disk requests. Using larger disk requests increases the time for each round, which increases the time required to start streaming a new video.

Buffer space. Although memory sizes continue to grow, they remain finite. Larger disk requests stress memory resources in two ways. For reads, larger disk requests are usually created by fetching

more data farther in advance of the actual need for it; this prefetched data must be buffered until it is needed. For writes, larger disk requests are usually created by holding more data in a write-back cache until enough contiguous data is dirty; this dirty data must be buffered until it is written to disk. The persistence problem discussed above can be addressed with non-volatile RAM, but the buffer space issue will remain.

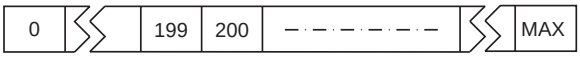
Irregular access patterns. Large disk requests are most easily generated when applications use regular access patterns and large files. Although sequential full-file access is relatively common [15, 120, 170], most data objects are much smaller than the disk request sizes needed to achieve good disk efficiency. For example, most files are well below 32 KB in size in UNIX-like systems [60, 155] and below 64 KB in Microsoft Windows systems [45, 170]. Directories and file attribute structures are almost always much smaller. To achieve sufficiently large disk requests in such environments, access patterns across data objects must be predicted at on-disk layout time. Although approaches to grouping small data objects have been explored [55, 60, 66, 134], all are based on imperfect heuristics, and thus they rarely group things perfectly. Even though disk efficiency is higher, misgrouped data objects result in wasted disk bandwidth and buffer memory, since some fetched objects will go unused. As the target request size grows, identifying sufficiently strong inter-relationships becomes more difficult.

Storage space management. Large disk requests are only possible when closely related data is collocated on the disk. Achieving this collocation requires that on-disk placement algorithms be able to find large regions of free space when needed. Also, when grouping multiple data objects, growth of individual data objects must be accommodated. All of these needs must be met with little or no information about future storage allocation and deallocation operations. Collectively, these facts create a complex storage management problem. Systems can address this problem with combinations of pre-allocation heuristics [21, 67], on-line reallocation actions [111, 134, 157], and idle-time reorganization [17, 114]. There is no straightforward solution and the difficulty grows with the target disk request size, because more related data must be clustered.

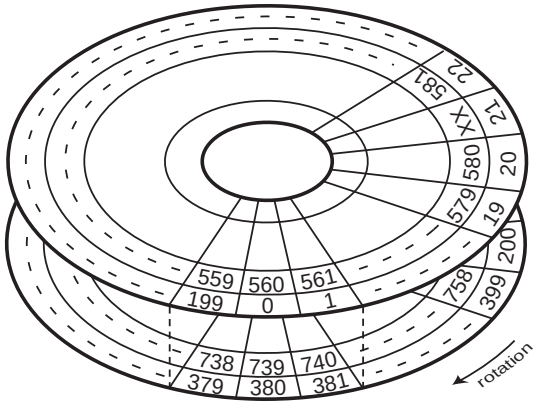
B.2 Understanding two efficiency-impacting disk characteristics

In light of the previous discussion, an alternative operating system-level approach is needed that improves the performance of the disk subsystem without resorting to ever-larger request sizes. In this section we identify an approach—track-aligned, track-sized disk accesses—that uses knowledge of performance-impacting disk characteristics to improve the efficiency with which the disk responds to requests.

The first uses of disks in the 1950s ignored the effects of geometry in the interest of achieving a working system. Later, algorithms were developed that paid attention to disk geometry in order to improve disk efficiency. These algorithms were often hard-coded and hardware-specific, making them fragile across generations of hardware. To address this, a layer of abstraction was standardized between operating systems and disks, virtualizing disk storage as a flat array of fixed-sized blocks.



(a) System's view of storage



(b) Mapping of LBNs onto physical sectors

Figure B.2: Standard system view of disk storage and its mapping onto physical disk sectors. *(a) illustrates the linear sequence of logical blocks, often 512 bytes, that the standard disk protocols expose. (b) shows one example mapping of those logical block numbers (LBNs) onto the disk media. The depicted disk drive has 200 sectors per track, two media surfaces, and track skew of 20 sectors. Logical blocks are assigned to the outer track of the first surface, the outer track of the second surface, the second track of the first surface, and so on. The track skew accounts for the head switch delay to maximize streaming bandwidth. The picture also shows a defect between the sectors with LBNs 580 and 581, depicted as XX, which has been handled by slipping. Therefore, the first LBN on the following track is 599 instead of 600.*

Unfortunately, this abstraction hides too much information, making the OS's task of maximizing disk efficiency more difficult than necessary.

Modern storage protocols, such as SCSI and IDE/ATA, expose storage capacity as a linear array of fixed-sized blocks, as shown in Figure B.2(a). By building atop this abstraction, OS software need not concern itself with complex device-specific details, and code can be reused across the large set of storage devices that use these interfaces (e.g., disk drives and disk arrays). Likewise, by exposing only this abstract interface, storage device vendors are free to modify and enhance their internal implementations. Behind this interface, the storage device must translate the logical block numbers (LBNs) to physical storage locations. Figure B.2(b) illustrates this translation for a disk drive, wherein LBNs are assigned sequentially on each track before moving to the next. Disk drive advances over the past decade have conspired to make the track a sweet-spot for disk efficiency, yielding the 50% increase at Point B of Figure B.3, as discussed below.

Head switch. A head switch occurs when a single request accesses a sequence of LBNs whose on-disk locations span two tracks. This head switch consists of turning on the electronics for the appropriate read/write head and adjusting its position to account for inter-surface alignment imperfections. The latter step requires the disk to read servo information to determine the head's location

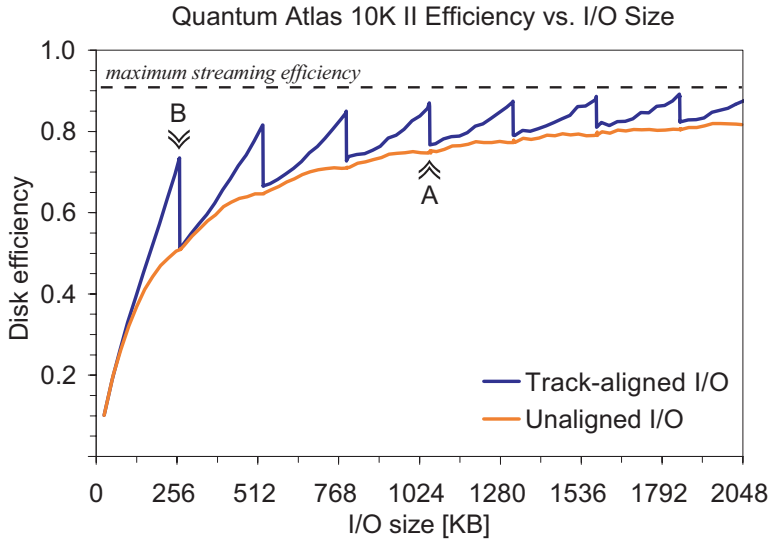


Figure B.3: Measured advantage of track-aligned access over unaligned access. *This graph extends the data in Figure B.1 with an additional cure representing track-aligned accesses. Both the "Track-aligned I/O" and "Unaligned I/O" curves show the disk efficiency for random, constant-sized reads within a Quantum Atlas 10K II's first zone (264 KB per track). Point B highlights the higher efficiency of track-aligned access (0.73, or 82% of the maximum) over unaligned access for a track-sized request. Point A shows where unaligned I/O efficiency catches up to the track-aligned efficiency at Point B. The peaks in the track-aligned curve correspond to multiples of the track size.*

and then to shift the head towards the center of the second track. In the example of Figure B.2(b), head switches occur between LBNs 199 and 200, 399 and 400, and 598 and 599.

Even compared to other disk characteristics, head switch time has improved little in the past decade. While disk rotation speeds have improved by 3× and average seek times by 2.5×, head switch times have decreased by only 20–40% (see Table B.1). At 0.6–1.1 ms, a head switch now takes about 1/5 of a revolution for a 15,000 RPM disk. This trend has increased the significance of head switches. Further, this trend is expected to continue, because rapid decreases in inter-track spacing require increasingly precise head positioning.

Naturally, not all requests span track boundaries. The probability of a head switch, P_{hs}, depends on workload and disk characteristics. For a request of N sectors and a track size of SPT sectors (sectors per track), $P_{hs} = (N-1)/SPT$, assuming that the requested locations are uncorrelated with track boundaries. For example, with 64 KB requests ($N = 128$) and an average track size of 192 KB ($SPT = 384$), a head switch occurs for every third access, on average. With N approaching SPT, almost every request will involve a head switch, which is why we refer to conventional systems as "track-unaligned" even though they are only "track-unaware". In this situation, track-aligned access improves the response time of most requests by the 0.6–1.1 ms head switch time.

Zero-latency access. A second disk feature that pushes for track-based access is zero-latency access, also known as immediate access or access-on-arrival. When disk firmware wants to read

Disk	Year	RPM	Head Switch	Avg. Seek	512B Sectors per Track	Number of Tracks	Capacity
HP C2247	1992	5400	1 ms	10 ms	96–56	25649	1 GB
Quantum Viking	1997	7200	1 ms	8.0 ms	216–126	49152	4.5 GB
IBM Ultrastar 18 ES	1998	7200	1.1 ms	7.6 ms	390–247	57090	9 GB
IBM Ultrastar 18LZX	1999	10000	0.8 ms	5.9 ms	382–195	116340	18 GB
Quantum Atlas 10K	1999	10000	0.8 ms	5.0 ms	334–224	60126	9 GB
Seagate Cheetah X15	2000	15000	0.8 ms	3.9 ms	386–286	103750	18 GB
Quantum Atlas 10K II	2000	10000	0.6 ms	4.7 ms	528–353	52014	9 GB

Table B.1: Trends in representative disk characteristics. *Note the minimal improvement in head switch time relative to improvements in other characteristics.*

N contiguous sectors, the simplest approach is to position the head (by a combination of seek and rotational latency) to the first sector and read the *N* sectors in ascending LBN order. With zero-latency access support, disk firmware can read the *N* sectors from the media into its buffers in any order. In the best case of reading exactly one track, the head can start reading data as soon as the seek is completed; no rotational latency is involved because all sectors on the track are needed. The *N* sectors are read into an intermediate buffer, assembled in ascending LBN order, and sent to the host. The same concept applies to writes, except that data must be moved from host memory to the disk's buffers before it can be written onto the media.

As an example of zero-latency access on the disk from Figure B.2(b), consider a read request for LBNs 200–399. First, the head is moved to the track containing these blocks. Suppose that, after the seek, the disk head is positioned above the sector containing LBN 380. A zero-latency disk can immediately read LBNs 380–399. It then reads the sectors with LBNs 200–379. In this way, the entire track can be read in only one rotation even though the head arrived in the "middle" of the track.

The expected rotational latency for a zero-latency disk decreases as the request size increases, as shown in Figure B.4. Therefore, a request to the zero-latency access disk for all *SPT* sectors on a track requires only one revolution after the seek. An ordinary disk, on the other hand, has an expected rotational latency of $(SPT-1)/(2\cdot SPT)$, or approximately 1/2 revolution, regardless of the request size and thus a request requires anywhere from one to two (average of 1.5) revolutions.

Putting it all together. For requests around the track size (100–500 KB), the potential benefit of track-based access is substantial. A track-unaligned access for *SPT* sectors involves four delays: seek, rotational latency, *SPT* sectors worth of media transfer, and head switch. An *SPT*-sector track-aligned access eliminates the rotational latency and head switch delays. This reduces access times for modern disks by 3–4 ms out of 9–12 ms, resulting in a 50% increase in efficiency.

Of course, the real benefit provided by track-based access depends on the workload. For example, a workload of random small requests, as characterizes transaction processing, will see minimal improvement because request sizes are too small. At the other end of the spectrum, a system that sequentially reads a single large file will also see little benefit, because positioning costs can be

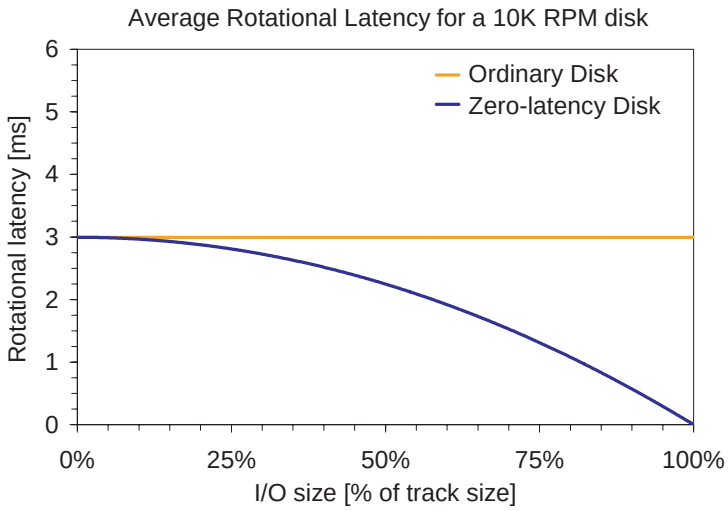


Figure B.4: Average rotational latency for ordinary and zero-latency disks as a function of track-aligned request size. *The request size is expressed as a percentage of the track size.*

amortized over megabyte sized transfers and the disk's prefetching logic will ensure that this occurs. Track-based access provides the highest benefit to applications with medium-sized I/Os. One set of examples is streaming media services, such as video servers, audio (MP3) servers, and content distribution network (CDN) caches. Another includes storage components (e.g., Network Appliance's filers [84], HP's AutoRAID [176], or EMC's Symmetrix) that map data to disk locations in mid-sized chunks. A third is a general-purpose operating system's memory management in reading and storing data between its internal buffer cache and attached storage devices. This is one of the paths we chose to explore in further detail, as described in the following subsection.

B.3 OS-level awareness of efficient disk access patterns

Track-based disk access is a design option for any system component that allocates disk locations and generates disk requests. In some systems, like the one used in our experiments, these decisions are made in the system software (e.g., file system) of a workstation, file server, or content-caching appliance. In others, the system software decisions are overridden by a logical disk [42] or a high-end disk array controller [159, 176], using some sort of mapping table to translate requested LBNs to internal disk locations. Track-based disk access is appropriate within any of these systems, and it requires relatively minor changes to existing systems. This section discusses practical design considerations involved with these changes.

To utilize track boundary information, the algorithms for on-disk placement and request generation must support variable-sized extents. Extent-based file systems, such as NTFS [117] and XFS [161], allocate disk space to files by specifying ranges of LBNs (extents) associated with each file. Such systems lend themselves naturally to track-based alignment of data: during allocation,

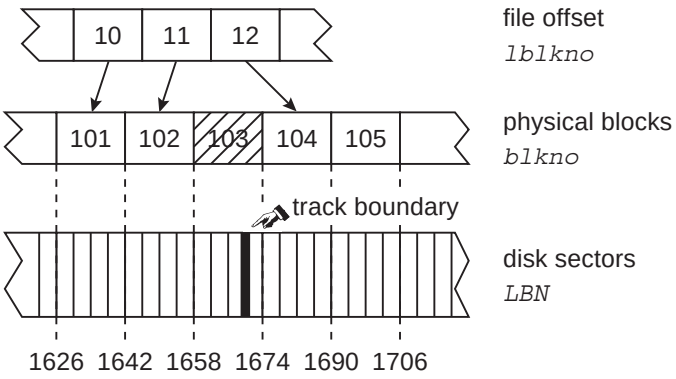


Figure B.5: An example mapping of operating system-level blocks to disk sectors for the FreeBSD Fast File System (FFS). *In this example, physical block 101 maps directly to disk sectors 1626–1641. Block 103 is treated as an "excluded" block in our implementation because it spans the disk track boundary between LBNs 1669–1670.*

extent ranges can be chosen to fit track boundaries. Block-based file systems, such as Ext2 [21] and FFS [116], group LBNs into fixed-size allocation units (blocks), typically 4 KB or 8 KB in size. Block-based systems can approximate track-sized extents by placing sequential runs of blocks such that they never span track boundaries. This approach wastes some space when track sizes are not evenly divisible by the block size. However, this space is usually less than 5% of total storage space and could be reclaimed by the system for storing inodes, superblocks, or fragmented blocks. Alternately, this space can be reclaimed if the cache manager can be modified to handle partially-valid and partially-dirty blocks.

We have developed a prototype implementation of a traxtent-aware file system in FreeBSD. This implementation identifies track boundaries and modifies the FreeBSD FFS implementation to take advantage of this information. The following three paragraphs describe the few, small changes required to integrate traxtent-awareness into FreeBSD FFS.

Excluded blocks and traxtent allocation. We introduce the concept of the *excluded* block, highlighted in Figure B.5. Blocks that span track boundaries are excluded from allocation decisions by marking them as used in the free-block map. Whenever the preferred block (the next sequential block) is excluded, we instead allocate the first block of the closest available traxtent. When possible, mid-size files are allocated such that they fit within a single traxtent. On average, one out of every twenty blocks of the Quantum Atlas 10K is excluded under our modified FFS. As per-track capacity grows, the frequency of excluded blocks decreases—for the Atlas 10K II, one in thirty is excluded.

Traxtent-sized access. No fundamental changes are necessary in the FFS clustered read-ahead algorithm. FFS properly identifies runs of blocks between excluded blocks as clusters and accesses them with a single disk request. Until non-sequential access is detected, we ignore the "sequential count" to prevent multiple partial accesses to a single traxtent; for non-sequential file sessions, the default mechanism is used. We handle the special case where there is no excluded block between

traxtents by ensuring that no read-ahead request goes beyond a track boundary. At a low level, unmodified FreeBSD already supports command queuing at the device and attempts to have at least one outstanding request for each active data stream.

Traxtent data structures. When the file system is created, track boundaries are identified, adjusted to the file system's partition, and stored on disk. At mount time, they are read into an extended FreeBSD `mount` structure. We chose the mount structure because it is available everywhere traxtent information is needed.

Our experimentation uncovered an additional design consideration: current systems only realize the full benefit of track-based requests when using command queueing at the disk. Although zero-latency disks can access LBNs on the media in any order, current SCSI and IDE/ATA controllers only allow for in-order delivery to or from the host. As a result, bus transfer overheads hide some of the benefit of zero-latency access. By having multiple requests outstanding at the disk, the next request's seek can be overlapped with the current request's bus transfer, yielding the full disk efficiency benefits shown in Figure B.3. Fortunately, most modern disks and most current operating systems support command queueing at the disk.

B.4 System-oriented evaluation using production-device experimentation

The experiments described in this section were performed on two disks that support zero-latency access (Quantum Atlas 10K and Quantum Atlas 10K II) and two disks that do not (Seagate Cheetah X15 and IBM Ultrastar 18 ES). The disks were attached to a 550 MHz Pentium III-based PC. The Atlas 10K II was attached via an Adaptec Ultra160 Wide SCSI adapter, the Atlas 10K and Ultrastar were attached via an 80 MB/s Ultra2 Wide SCSI adapter, and the Cheetah via a QLogic Fibre Channel adapter.

This section compares our prototype traxtent-aware FFS to unmodified FFS. We also include results for a modified FFS, here called *fast start* FFS, that aggressively prefetches contiguous blocks. The unmodified FFS slowly ramps up its prefetching as it observes sequential access to a file. The fast start FFS, on the other hand, prefetches up to 32 contiguous blocks on the first access to a file, thus approximating the behavior of the traxtent-aware FFS (albeit with larger requests and no knowledge of track boundaries).

Each test is performed on a freshly-booted system with a clean partition on a Quantum Atlas 10K. The tests verify the expected performance effects: small penalty for single sequential scan, substantial benefit for interleaved scans, and no effect on small file activity. We also identify and measure the worst-case scenario. The results are summarized in Table B.2.

Single large file. The first experiment is an I/O-bound linear scan through a 4 GB file. As expected, traxtent-FFS runs 5% slower than unmodified FFS or fast start FFS (199.8 s vs. 189.6 s and 188.9 s respectively). This is because FFS is optimized for large sequential single-file access and reads at the maximum disk streaming rate, whereas traxtent-FFS inserts an excluded block one out of every twenty blocks (5%). This penalty could be eliminated by changing the file system cache

	4 GB `scan`	512 MB `diff`	1 GB `copy`	PostMark	SSH build	`head *`
unmodified	189.6 s	69.7 s	156.9 s	53 trans/s	72.0 s	4.6 s
fast start	188.9 s	70.0 s	155.3 s	53 trans/s	71.5 s	5.5 s
traxtents	199.8 s	56.6 s	124.9 s	55 trans/s	71.5 s	5.2 s

Table B.2: Production-device experimentation results for the unmodified and modified FreeBSD Fast File System (FFS). *The FFS configuration for each run in this table is described in Section B.4. All but the* `head *` *values are an average of three runs. The individual run times deviate from their average by less than 1%. The* `head *` *value is an average of five runs and the individual runs deviate by less than 3.5%. Postmark reported the same number of transactions per second in all three runs for the respective FFS, except for one run of the unmodified FFS that reported 54 transactions per second.*

to support buffering of partial blocks (much like IP fragments) instead of using excluded blocks in large files; this approach would give the block-based system extent-like flexibility.

Multiple large files. The second experiment consists of the `diff` application comparing two large files. Because `diff` interleaves fetches from the two files, we expect to see a speedup from improved disk efficiency. For 512 MB files, traxtent-FFS completes 19% faster than unmodified FFS or fast start FFS. A more detailed analysis shows that traxtent-FFS performs 6724 I/Os (average size of 160 KB) in 56.6 s while unmodified FFS performs only 4108 I/Os (mostly 256 KB) but requires 69.7 s. The fast start FFS performs 4094 I/Os (all but one at 256 KB) and requires 70.0 s. Subtracting media transfer time, unmodified FFS incurs 6.9 ms of overhead (seek + rotational latency + track switch time) per request, and traxtent-FFS incurs only 2.2 ms of overhead per request. In fact, the 19% improvement in overall completion time corresponds to an improvement in disk efficiency of 23%, exactly matching the predicted difference between single-track accesses and 256 KB unaligned accesses on an Atlas 10K disk.

The third experiment verifies write performance by copying a 1 GB file to another file in the same directory. FFS commits dirty buffers as soon as a complete cluster is created, which results in two interleaved request streams to the disk. This test shows a 20% reduction in run time for traxtent-FFS over unmodified FFS (124.9 s vs. 156.9 s), and a similar reduction over fast start FFS, which finished in 155.3 s.

Small Files. Two application benchmarks are used to verify that the traxtent modifications do not penalize small file workloads. PostMark [96] simulates the small-file activity of busy Internet servers. Our experiments use PostMark v1.11 and its default parameters: 5–10KB files and 1:1 read-to-write and create-to-delete ratios. SSH-build [151] represents software development activity, replacing the Andrew benchmark. Its three phases unpack the compressed tar archive of SSH v1.2.27, generate the header files and Makefiles, and build the program executable.

As expected, we observe little difference. The SSH-build results differ by less than 0.2%, because the file system activity is dominated by small synchronous writes and cache hits. The fast start FFS performs exactly like the traxtent FFS having an edge of 0.2% over the unmodified FFS. PostMark is 4% faster with traxtents (55 transactions/second versus 53 for both unmodified and fast

start FFS), because the few track switches are avoided. Fast start is not important for PostMark, because the files consist of only 1–3 blocks.

One might view these results as a negative indication of traxtents' value, but they are not. Recall that FreeBSD FFS does not explicitly group small files into large disk requests. Such grouping has been shown to yield 2–8× throughput increases for static web servers [95], web proxy caches [152], and software development activities [60]. Based on our measurements, we expect that the additional 50% increase in throughput from traxtents would be realized given such grouping.

Worst case scenario. As expected, we observe no penalty to small file I/O and a minimal (5%) penalty to the unoptimized single stream case. For random file I/O, FFS's "sequential count" prefetch control replaces the traxtent-based fetch mechanism, preventing useless full-track reads. The one remaining worst-case scenario would be single-block reads to the beginnings of many large files; in this case, the original FFS will fetch the first 8KB block and prefetch the second, whereas the modified FFS will fetch the entire first traxtent (≈ 160 KB). To evaluate this scenario, we ran an experiment, called `head *`, that reads the first byte of 1000 200 KB files. The results show a 45% penalty for traxtents (3.6 s vs. 5.2 s), closely matching the predicted per-request service time difference (5.6 ms vs. 8.0 ms). Fortunately, this scenario is not often expected to arise in practice. Not surprisingly, the fast start FFS performs even worse than the traxtent FFS with an average runtime of 5.5 s as it prefetches even more unnecessary data.

B.5 The need for timing-accurate storage emulation

The work described herein analyzes the system-level integration of track-aligned extents (*traxtents*), logical block extents that are aligned and sized by the operating system so as to match the corresponding disk track size. By utilizing this awareness of a small amount of disk-specific knowledge, an operating system can significantly increase the efficiency of mid-to-large-sized requests—i.e., request sizes of 100 KB and up. Traxtent-aware access results in up to 50% higher disk efficiency—the fraction of total access time spent moving data to or from the media—which translates into up to 20% improvements in application-level performance in our experiments. The demonstrated application-level improvement stems from two main sources. First, ensuring track-aligned access by the operating system minimizes the number of track switches, whose times have not decreased much over the years and are now significant (0.6–1.1 ms) relative to other delays. Second, ensuring full-track accesses by the operating system eliminates rotational latency (averaging 3 ms per request at 10,000 RPM) for disk drives whose firmware supports zero-latency access.

The results in this section were obtained using the technique of production-device experimentation; this choice was mandated because no other full-system evaluation approach for disks was available when the analysis was performed. Although the experiments were highly successful and the results well-received externally, the necessity of production device-based experimentation resulted in two primary limitations on future evaluations. First, zero-latency access is not a supported feature of all current disk products. A product developer who is interested in evaluating the potential

of track-aligned extents in a modified disk has no experimental options short of building a prototype with the requisite firmware functionality. Second, researchers cannot look at the performance impact of hypothetical device modifications in a full-systems context—i.e., whether considering physical changes such as head-switch time reduction or software changes such as a new scheduling algorithm—without again constructing a physical prototype. For example, one promising approach that we were unable to pursue is support for out-of-order bus delivery in device firmware, which we expect would further improve the efficiency of traxtent-based accesses. The experimental approach and results described in this section would be equally obtainable using a timing-accurate storage emulator built upon a validated device model, with the added advantage that future experimentation would not share these limitations.

APPENDIX C
FINE-GRAINED AND EFFICIENT TIMEPOINT DETERMINATION

Because time stamps are taken multiple times in several locations during each storage request (as discussed in Section 5.2), and taking into account the possible sub-millisecond resolution of an emulated storage request, the facility for taking time stamps must be both efficient and have a small resolution. One resource that meets these goals of efficiency and resolution is the time stamp counter (TSC) register on the Pentium 4 processor used in our architecture. This appendix describes the technique we used to calibrate the clocks on the host and emulation system using the TSC register.

C.1 Using a processor cycle counter to measure elapsed time

The TSC is a 64-bit cycle counter register, reset to zero at processor power-up or reset, that is incremented once per processor clock cycle. Effective use of this counter requires determining the number of TSC cycles that elapse per second (CPS). CPS can be determined empirically by counting the number of cycles that elapse over a known time interval, beginning at an initial time t_i and ending at the final time t_f, as shown in Equation C.1:

$$\text{CPS} = \frac{(\text{cycle count at } t_f) - (\text{cycle count at } t_i)}{(\text{system time at } t_f) - (\text{system time at } t_i)} \tag{C.1}$$

CPS is a hardware-dependent value and will likely vary between the host and target systems. It should not be necessary to recalculate CPS after an initial value has been determined.

Calculations of the elapsed time in milliseconds (ΔT) are used frequently during an experiment. The emulator needs to know the elapsed time since the beginning of an experiment when a request arrives, and it needs to determine the elapsed time since the beginning of a currently active request. ΔT is calculated during an experiment by comparing the TSC at an initial time t_i' and again at the current time t', and converting these values to milliseconds as shown in Equation C.2:

$$\Delta T = \frac{(\text{cycle count at } t') - (\text{cycle count at } t_i')}{\text{CPS}} \times 1000.0 \tag{C.2}$$

In a remote emulation environment, where the host and target systems utilize different physical hardware, it is necessary to verify that the calculated rates of passage of time on both systems are consistent. It is not necessary for the clocks to be synchronized absolutely with respect to each other, or for the clock cycles to advance at identical rates; rather, externally-driven events that propagate end-to-end through the storage request path (such as request arrivals) can be used as relative synchronization points for such analysis. For example, this verification can be achieved by comparing the request interarrival times as measured at different points throughout the system: even

if there is a large difference between the measured length of time *of* a request on the host and on the emulator, then assuming no variance in propagation delays there should be no difference between the measured time *between* request arrivals. This concept is demonstrated graphically in Figure C.1. Experimental verification of the correspondence of these times is described in the following section.

C.2 Clock calibration on the host and emulation systems

The basis of time stamp measurements in our implementation of a timing-accurate storage emulator involves measuring elapsed time both in terms of processor cycles and actual wall clock time. Processor cycles are measured using the Pentium 4 RDTSC (Read Time Stamp Counter) assembly-language operation. This operation was invoked as shown in Figure C.2. System times were obtained using the `gettimeofday()` POSIX-compliant system call, which returns the current time expressed as the number of seconds and microseconds since a fixed time in the past.[C.1] Use of RDTSC is preferable over the `gettimeofday()` system call because of both the low resolution and high overhead of the system call.

To improve the accuracy of the `gettimeofday()` system call, we enabled the Network Time Protocol (NTP) service on both the host and emulator systems and each was synchronized with a pair of Public NTP Secondary (stratum 2) Time Servers. The time-update daemon running on each system microadjusts the system clock (i.e., the value returned by `gettimeofday()`) based on the daemon's synchronization with the network time servers. However, the daemon does not modify the values returned by the TSC. This has the interesting effect of causing a slight drift in the CPS measured via Equation C.1, as shown in Figure C.3 and Figure C.4. These results suggest that time intervals of as short as 1 second provide estimates for the current CPS on our 2 GHz systems that are accurate within one part in 10,000, although in practice intervals of 50 s or greater provide the best precision. When using a clock-adjusting scheme such as the NTP daemon, an evaluator could mitigate the drift effect by calculating CPS in this manner before every experiment.

When operating in a remote emulation environment, it is necessary to verify that the measured values for CPS on both the host and emulator systems produce timelines with identical rates of passage of time on each system, as shown in Figure C.1. The results are shown in Table C.1. The errors are negligible, which yields confidence in the accuracy with which the CPS values have been determined. To demonstrate an erroneous determination, we intentionally introduced a 10% error into the CPS value for the emulated system and repeated this evaluation. The erroneous results are shown in Table C.2; the magnitude of the error indicates that the determination is invalid.

[C.1] The number of seconds returned is a signed 32-bit integer that represents the differential number of seconds compared with a fixed time. The fixed time is currently universally defined as the dawn of the modern computing age: the stroke of midnight on January 1, 1970. At the time of this writing, it is unclear whether the computing age will come to an abrupt halt when this 32-bit signed integer overflows on January 18, 2038. However, the author fully believes that the principles of timing-accurate storage emulation will remain valid at the transition and thereafter. Additionally, the author finds it curiously fascinating to think of what sorts of storage devices will be being emulated in three decades.

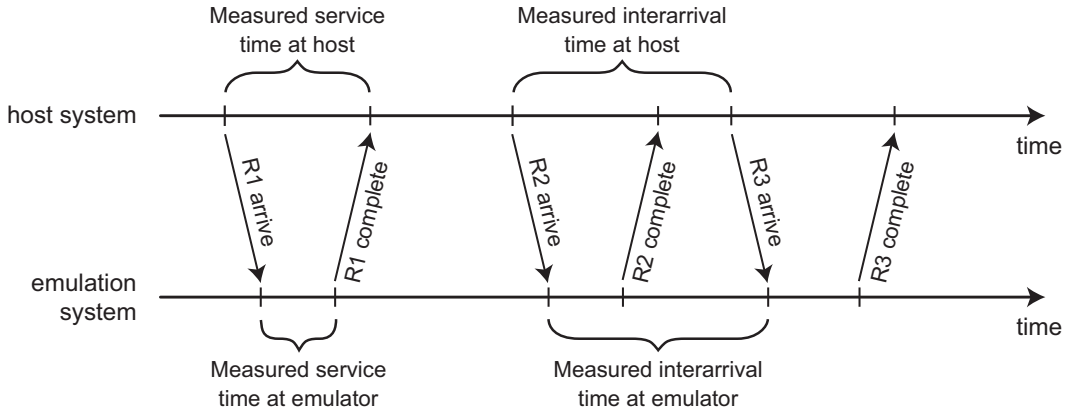


Figure C.1: Verifying that unsynchronized clocks running on separate hardware systems are advancing at the same rate of time. *R1, R2, and R3 represent three temporally sequential requests. Although the measured service times of individual requests might differ on the host and emulation systems due to delays along the request propagation path, the measured request interarrival times (or intercompletion times) should be equal as long as the clocks on both systems are advancing at equal rates.*

```
inline void time_stamp (u_int64_t *d) {
    asm volatile ("rdtsc" : "=&A" (*d));
}

void foo (void) {
    u_int64_t ts;
    time_stamp(&ts);
}
```

Figure C.2: Invocation of the RDTSC assembly-language operation in a C-language program. *The operation returns the number of processor cycles elapsed since processor power-on or reset. This is an efficient and precise method of obtaining timestamps during an experiment.*

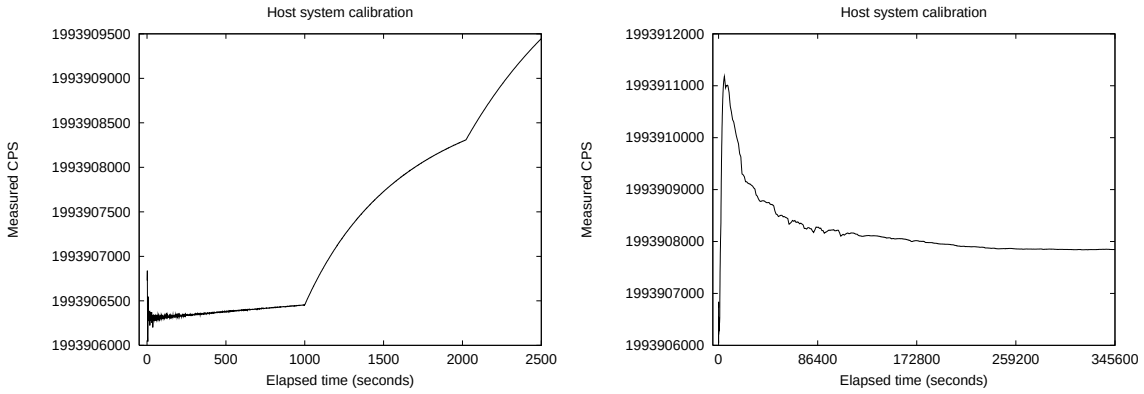


Figure C.3: Drifting of the measured processor cycles-per-second (CPS) for the host system. *Note that the Y-axis does not begin at the origin in these graphs. This graph was generated according to Equation C.1 by recording an initial cycle count and system time at time t_i, then calculating a new CPS value using that t_i once per second over a period of four days (345,600 s); the measurement at (for example) $t_f = 1000$ was calculated 1000 s after t_i using the current cycle count and system time. The observed drift is caused by the network time protocol daemon, which effects variable microadjustments of the system time approximately every 1000 seconds in order to keep the system time synchronized with that of an external server. This causes the system's definition of "second" to change over time (the daemon does not affect the processor), causing the CPS value to vary from the four-day mean of 1,993,908,205 cycles by -0.00011% to +0.00015%. The left-hand graph is a domain-limited reproduction of the right-hand graph over the range [0,2500]; the first point at which the microadjustment changes occurs at time 1001.*

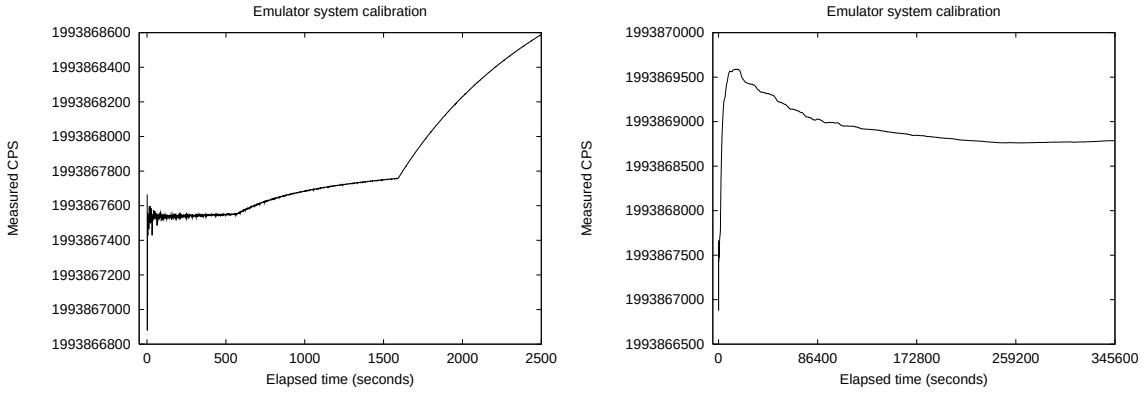


Figure C.4: Drifting of the measured processor cycles-per-second for the emulator system. *Note that the Y-axis does not begin at the origin in these graphs. This represents is the same measurement as in Figure C.3, except the data were taken on the emulation system. The four-day mean is 1,993,868,931; the drift ranges from -0.00010% to +0.000033%. The left-hand graph is a domain-limited reproduction of the right-hand graph over the range [0,2500]; the first point at which the microadjustment changes occurs at time 569.*

	QLogic		Adaptec	
	value (ms)	*percent*	*value (ms)*	*percent*
Interarrival time average error	0.000025	0.000025%	0.000043	0.000043%
Interarrival time RMS error	0.29	0.29%	0.20	0.20%
Overall error (based on arrival times)	0.25	0.000025%	0.43	0.000043%
Intercompletion time average error	0.000025	0.000025%	0.000066	0.000066%
Intercompletion time RMS error	0.035	0.035%	0.037	0.037%
Overall error (based on completion times)	0.25	0.000025%	0.41	0.000041%

Table C.1: Verification of the measured CPS values in a remote emulation environment. *This represents a simple workload that repeatedly sends an identical small read request (block offset zero, length 2 blocks) to the emulated device with frequency 10 Hz (i.e., an average 100 ms inter-arrival time), over the span of N=10,000 requests. This was performed for emulated devices using both the QLogic and Adaptec host bus adapters. We examined the average and root-mean-square interarrival time difference for adjacent request pairs (i.e., the $n-1$ times between the arrival of request n and request $n+1$, where $1 \leq n \leq (N-1)$), as well as the overall error between the first and final requests (based on the arrival time of request 1 and request N). This analysis was repeated using the intercompletion times of the requests.*

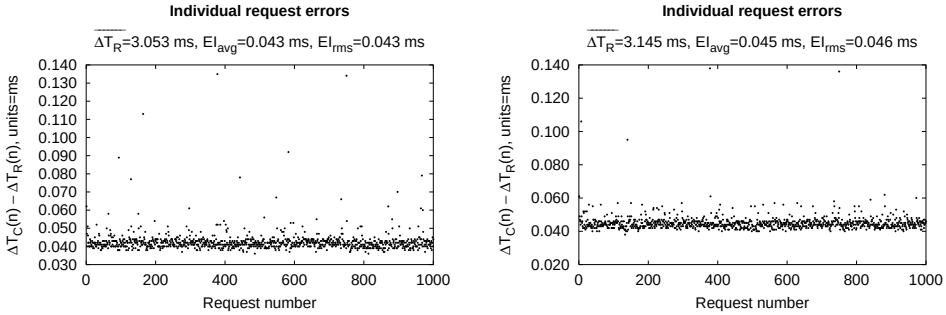
	QLogic		Adaptec	
	value (ms)	*percent*	*value (ms)*	*percent*
Interarrival time average error	9.1	9.1%	9.1	9.1%
Interarrival time RMS error	9.1	9.1%	9.1	9.1%
Overall error (based on arrival times)	91000	9.1%	91000	9.1%
Intercompletion time average error	9.1	9.1%	18	18%
Intercompletion time RMS error	9.1	9.1%	13	13%
Overall error (based on completion times)	91000	9.1%	91000	9.1%

Table C.2: The effect of erroneous CPS values at verification time. *This is the same type of evaluation as shown in Table C.1, except here the emulator system was artificially calibrated with an erroneous CPS value 10% too large. Observation of this mismatch would suggest that any experimental results based on this calibration should be discarded as nonrepresentative of the emulated device model.*

APPENDIX D
FULL DATA FROM THE TIMING-ORIENTED EXPERIMENTATION

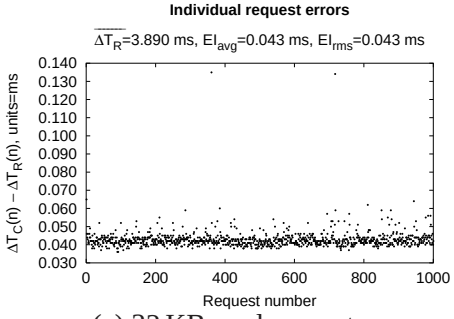
Many of the graphs in the figures from Chapter 5 and Chapter 6 are presented with limited X-axis range and Y-axis domain to simplify comparison among figures. The full plots of the individual request error subgraphs (i.e., the upper-left subgraph in the figures) are reproduced in this section. Note that this causes the Y-axis range to vary substantially among each of the subgraphs, though the X-axis domain is constant among all subgraphs in each figure. The plots for the experiments in Chapter 5 include:

The plots for the experiments in Chapter 6 include graphs representing the metrics for the Emulated Cheetah, Emulated 50K RPM, and Emulated MEMS models:

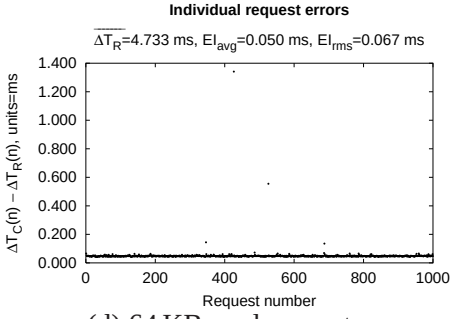


(a) 1 KB read requests

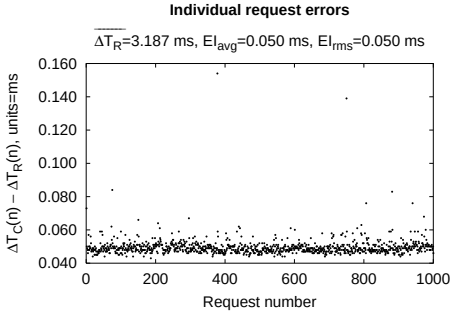
(b) 4 KB read requests



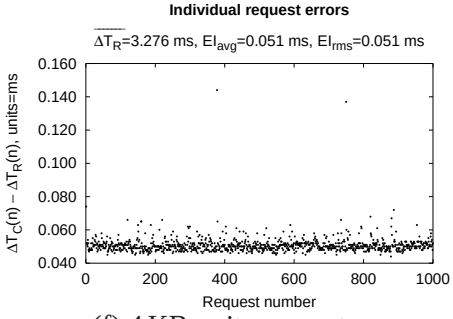
(c) 32 KB read requests



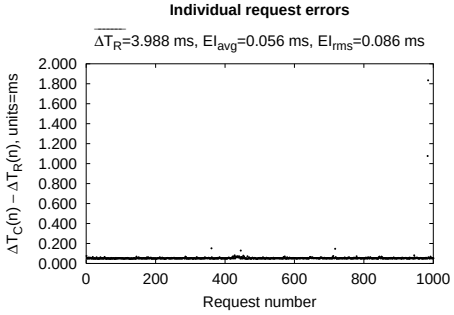
(d) 64 KB read requests



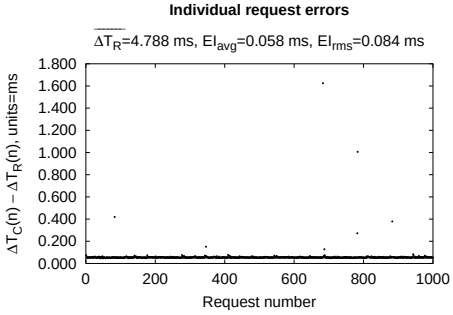
(e) 1 KB write requests



(f) 4 KB write requests



(g) 32 KB write requests



(h) 64 KB write requests

Figure D.1: $E_{2\rightarrow3}$**:** $\Delta T_{lookahead}$**=0** μ**s,** ΔT_{skew}**=0** μ**s, Adaptec.** *Note that the Y-axis range may vary substantially among each individual subgraph in this chapter.*

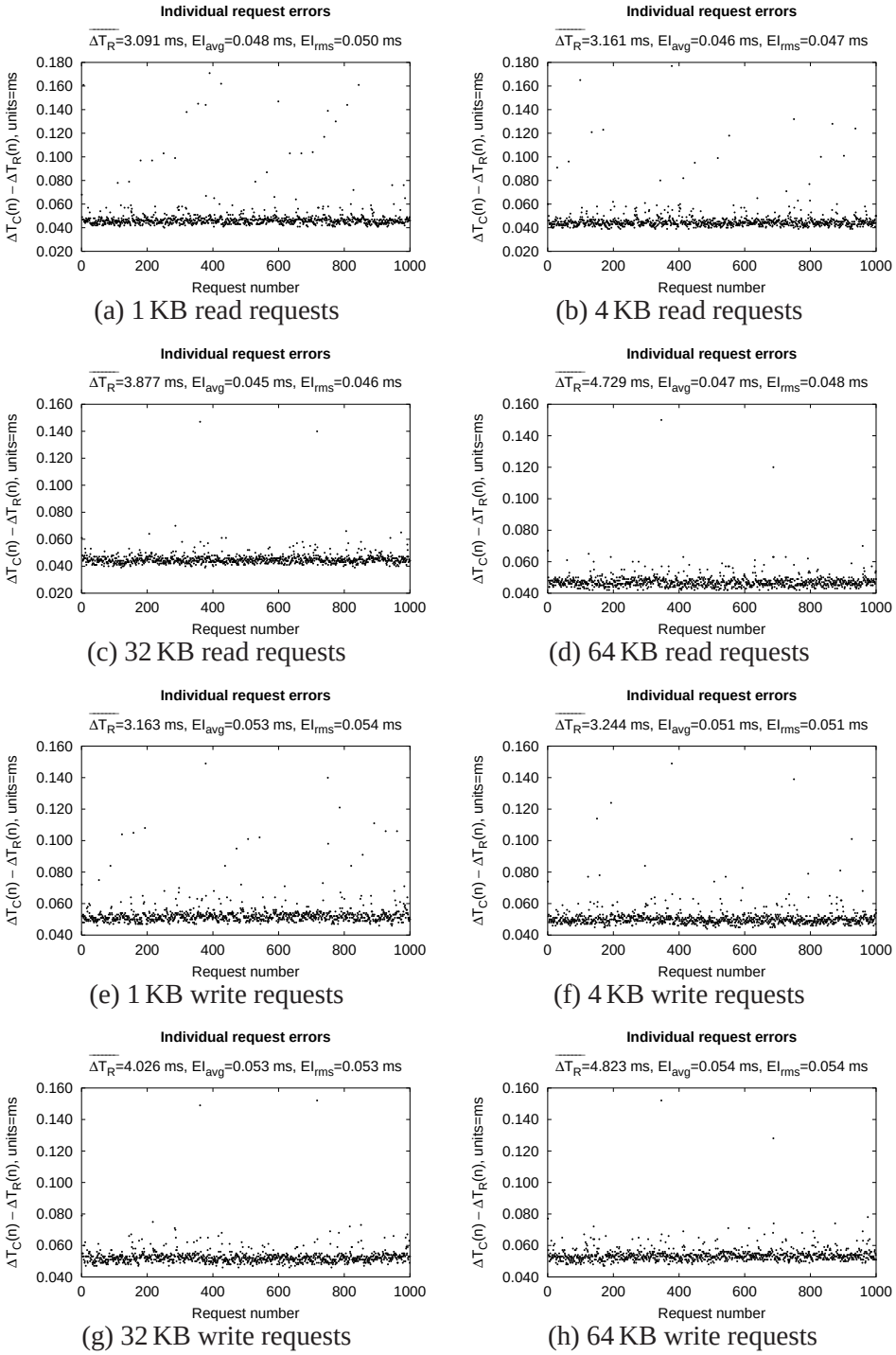


Figure D.2: $E_{2\rightarrow3}$**:** $\Delta T_{lookahead}$**=0** μ**s,** ΔT_{skew}**=0** μ**s, QLogic.**

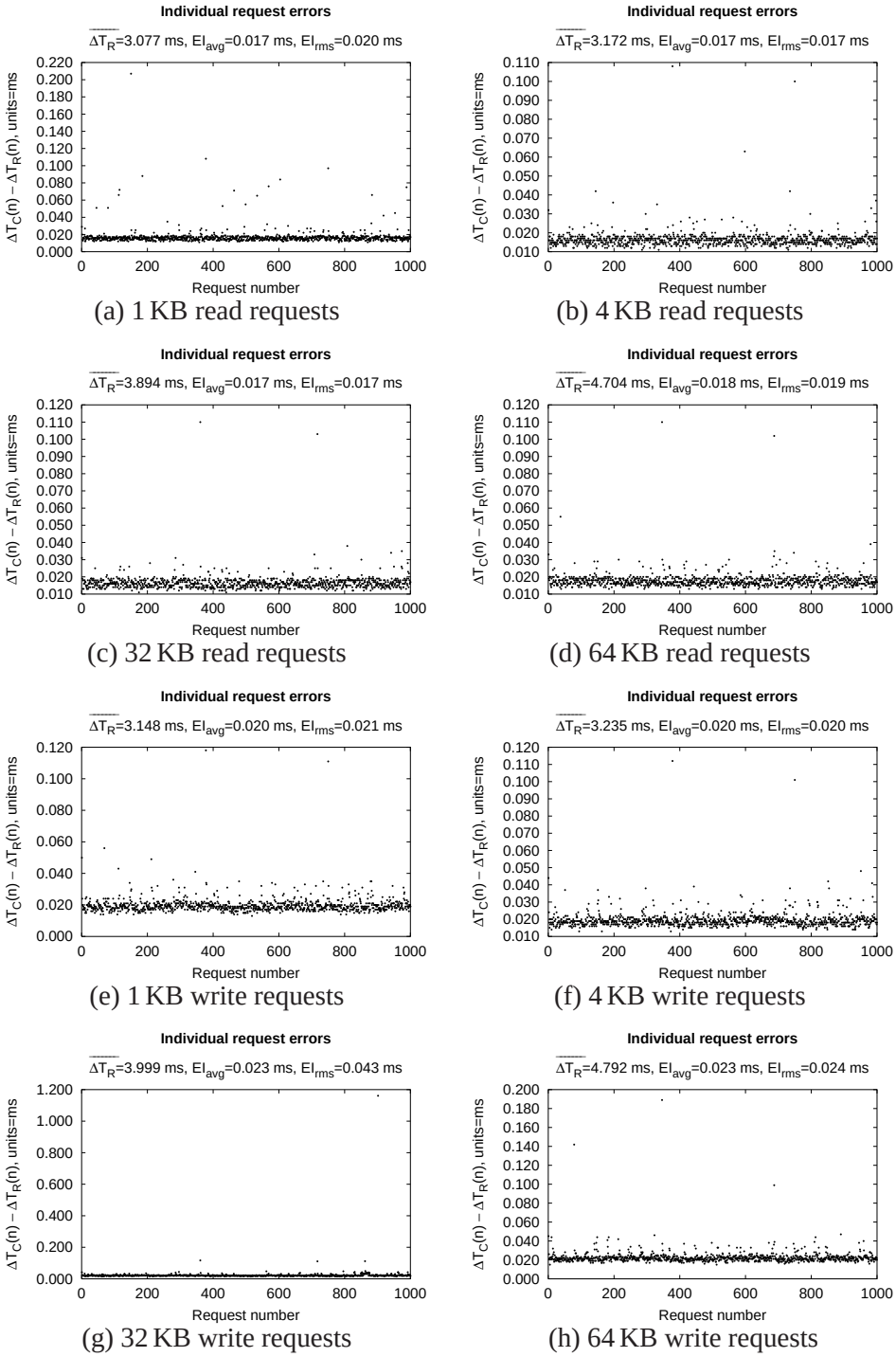


Figure D.3: $E_{2\rightarrow3}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=0** μ**s, Adaptec.**

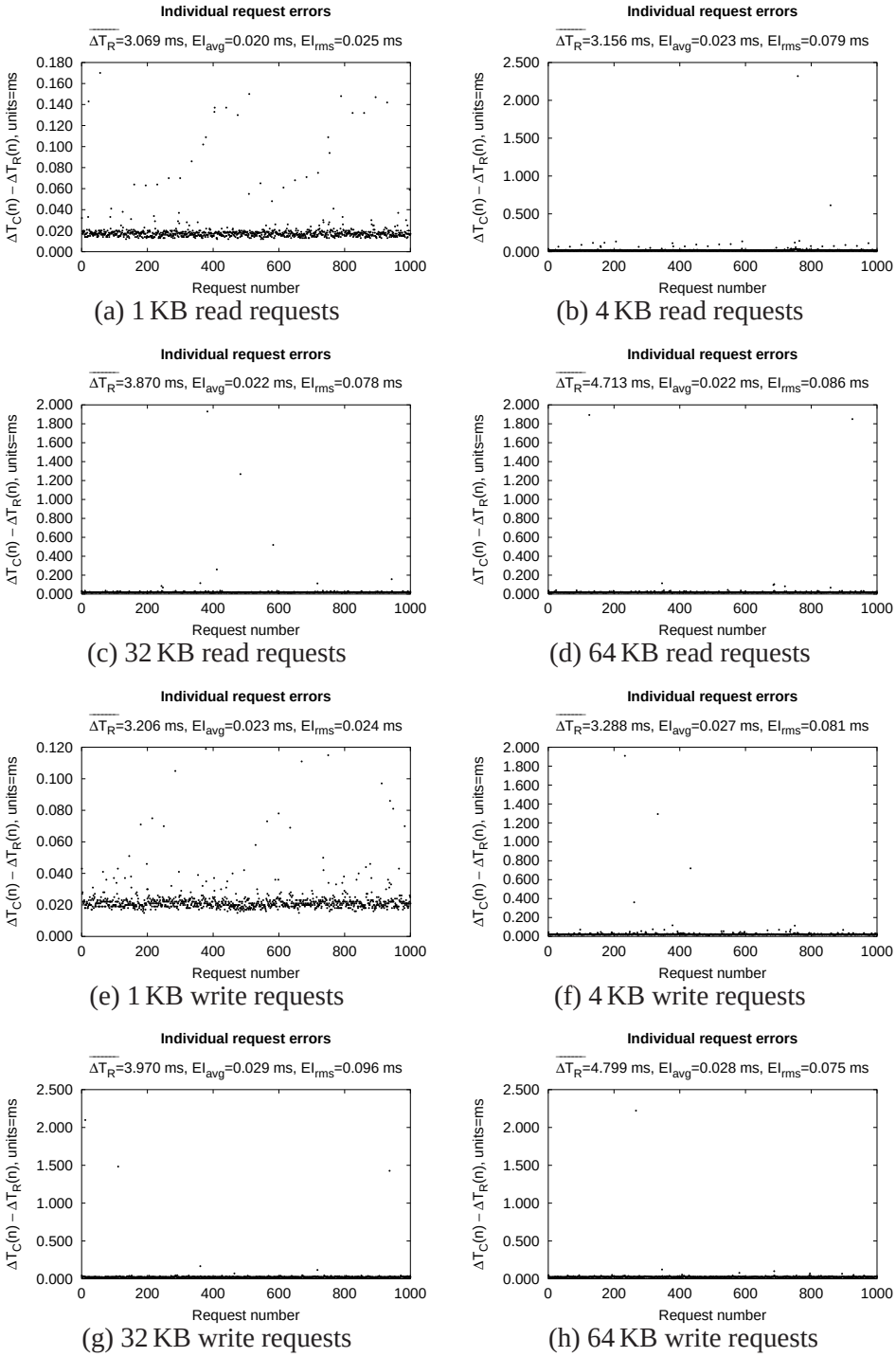
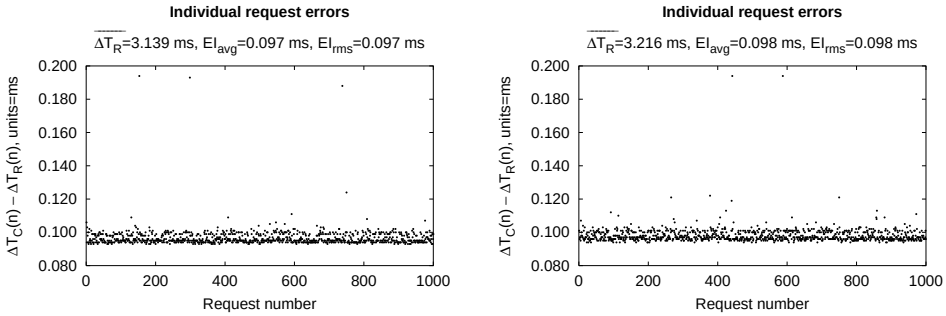
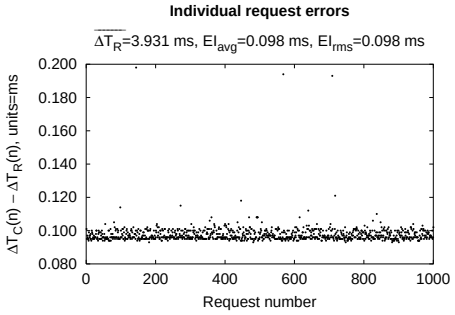


Figure D.4: $E_{2\rightarrow3}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=0** μ**s, QLogic.**

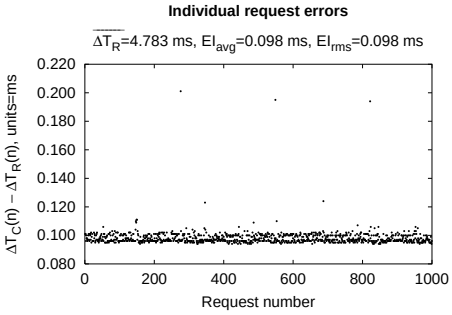


(a) 1 KB read requests

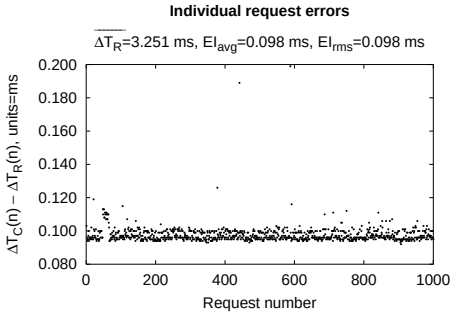
(b) 4 KB read requests



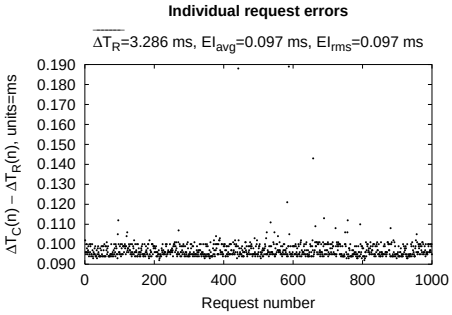
(c) 32 KB read requests



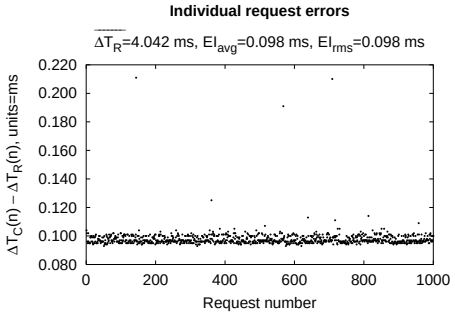
(d) 64 KB read requests



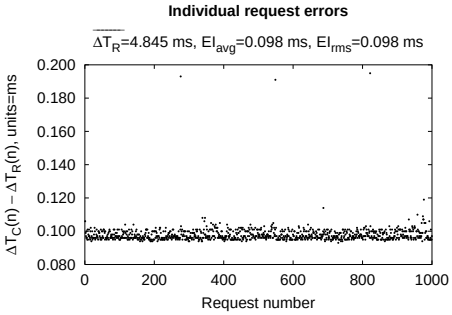
(e) 1 KB write requests



(f) 4 KB write requests



(g) 32 KB write requests



(h) 64 KB write requests

Figure D.5: $E_{1\rightarrow 2}$**:** $\Delta T_{lookahead}$**=0** μ**s,** ΔT_{skew}**=0** μ**s, Adaptec.**

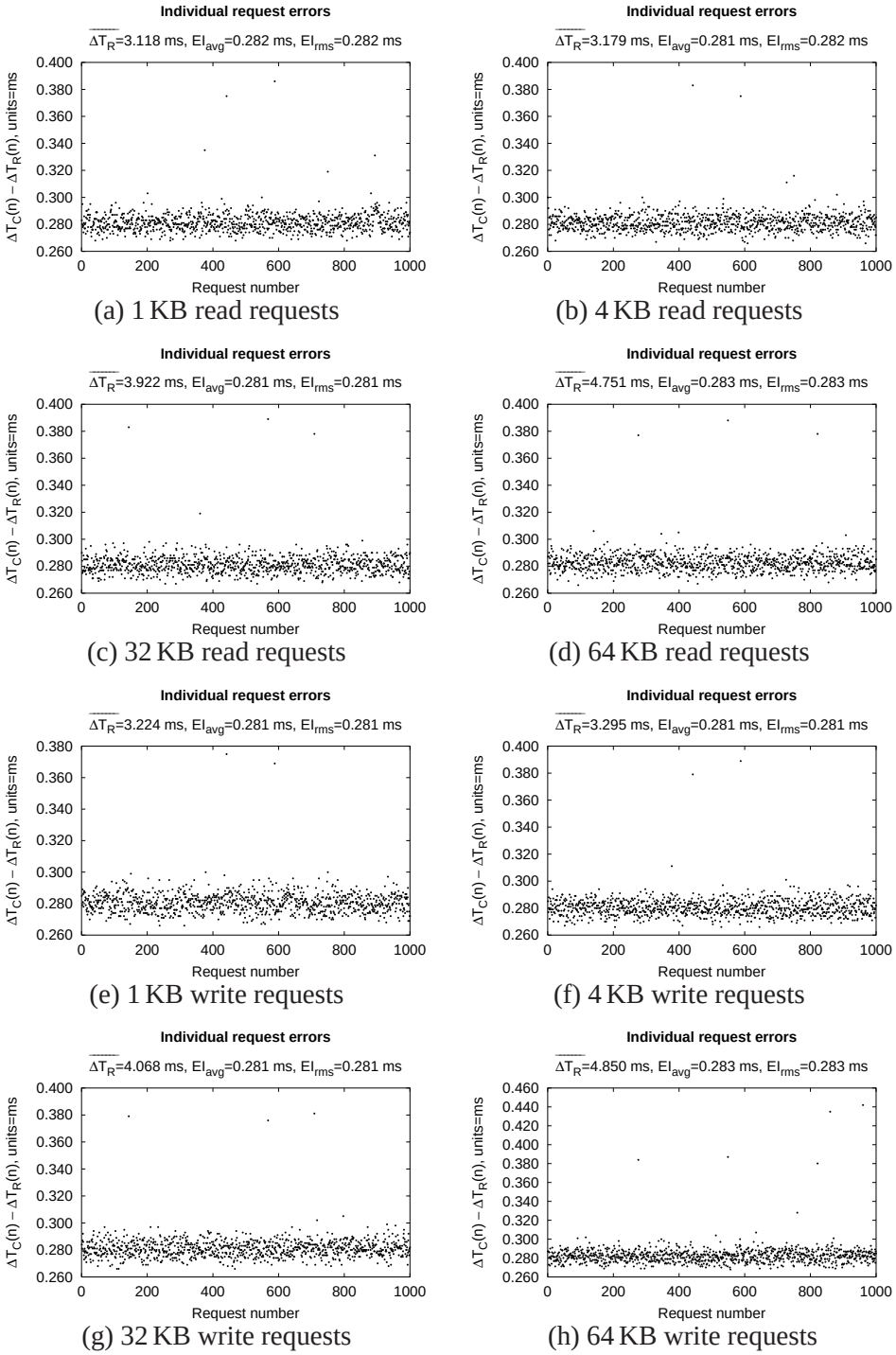
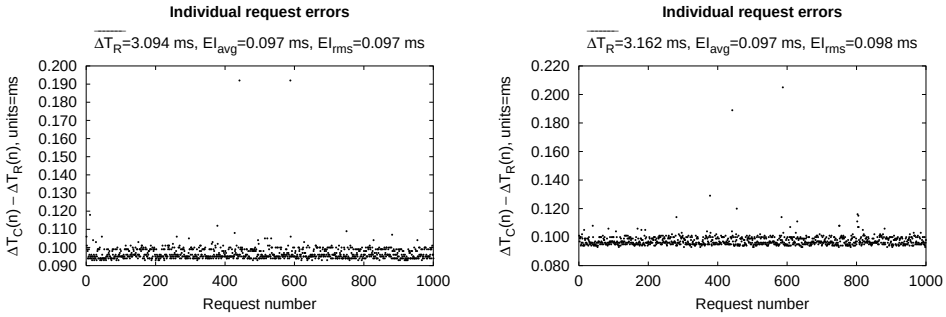
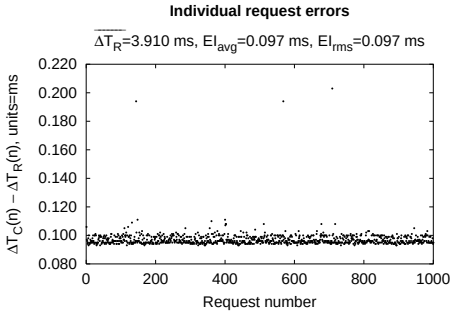


Figure D.6: $E_{1\rightarrow 2}$**:** $\Delta T_{lookahead}$**=0** μ**s,** ΔT_{skew}**=0** μ**s, QLogic.**

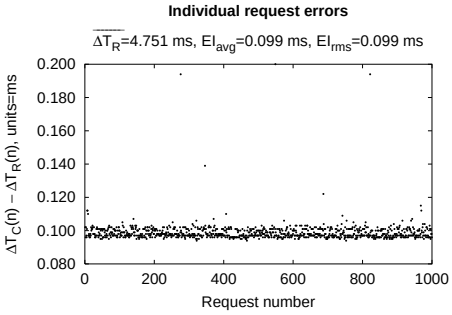


(a) 1 KB read requests

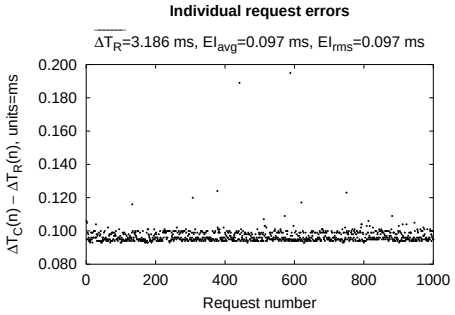
(b) 4 KB read requests



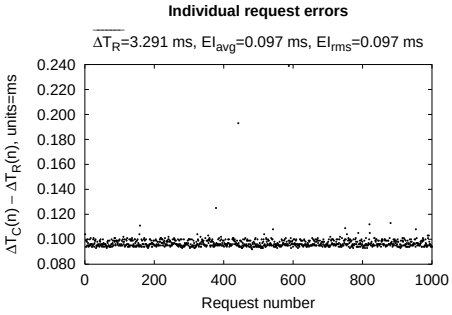
(c) 32 KB read requests



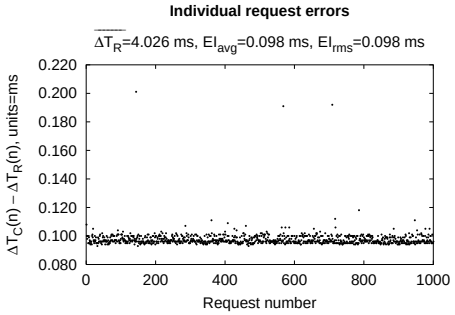
(d) 64 KB read requests



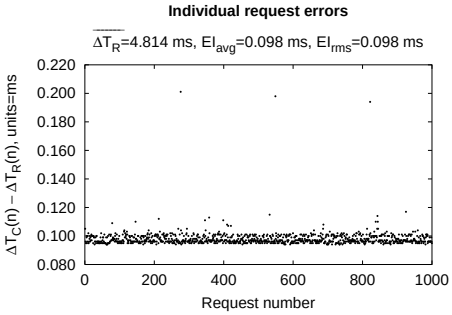
(e) 1 KB write requests



(f) 4 KB write requests

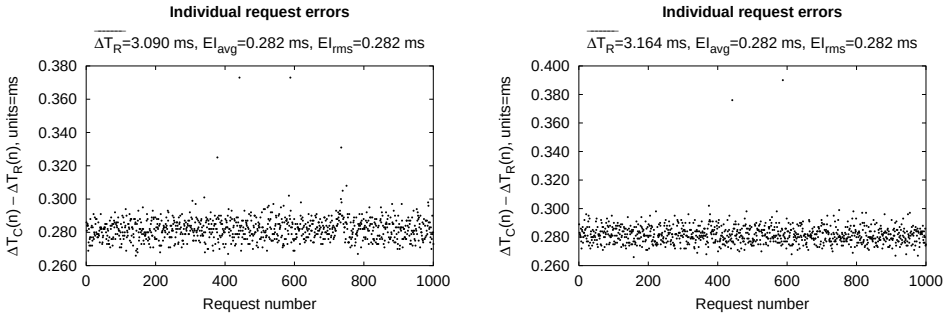


(g) 32 KB write requests



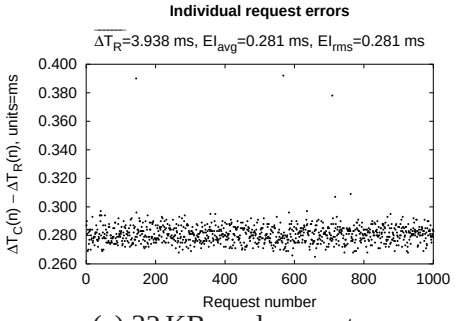
(h) 64 KB write requests

Figure D.7: $E_{1\rightarrow 2}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=0** μ**s, Adaptec.**

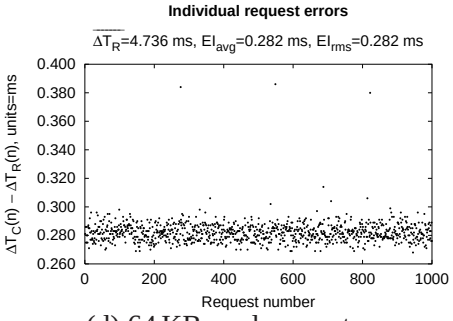


(a) 1 KB read requests

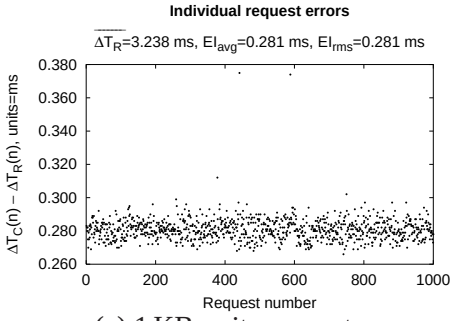
(b) 4 KB read requests



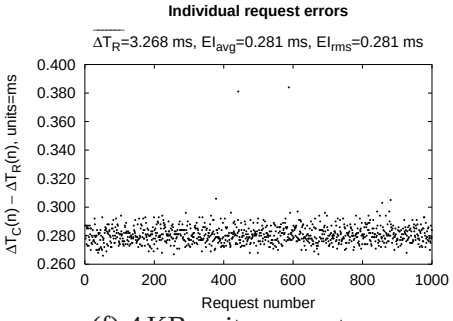
(c) 32 KB read requests



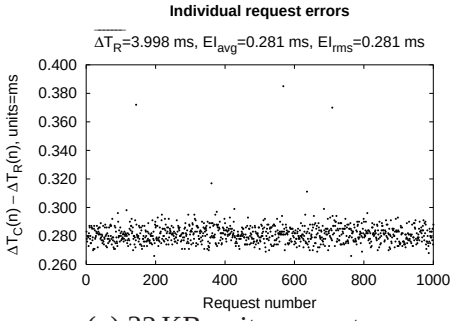
(d) 64 KB read requests



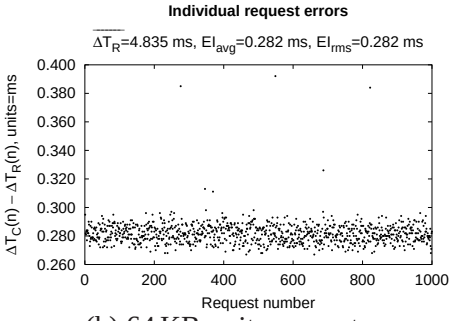
(e) 1 KB write requests



(f) 4 KB write requests

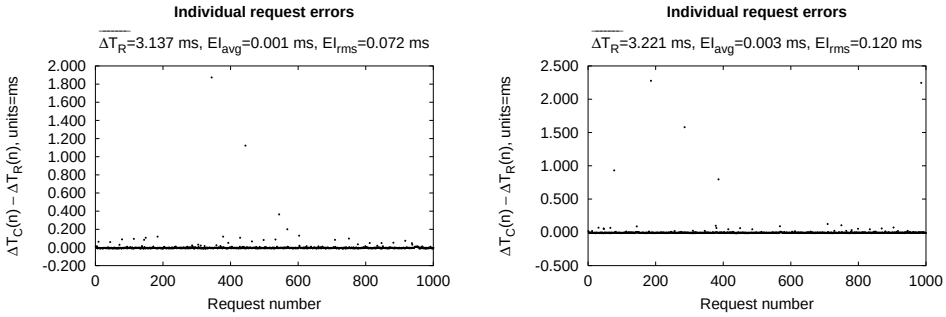


(g) 32 KB write requests



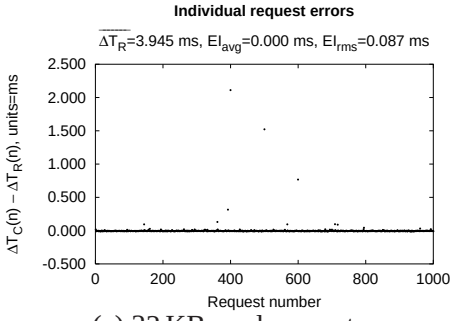
(h) 64 KB write requests

Figure D.8: $E_{1\rightarrow 2}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=0** μ**s, QLogic.**

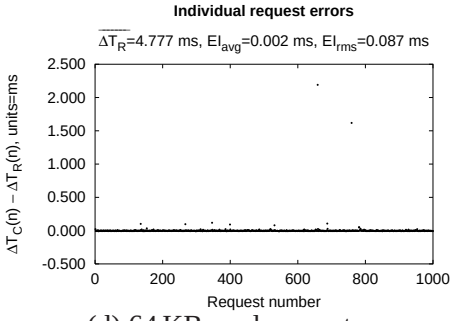


(a) 1 KB read requests

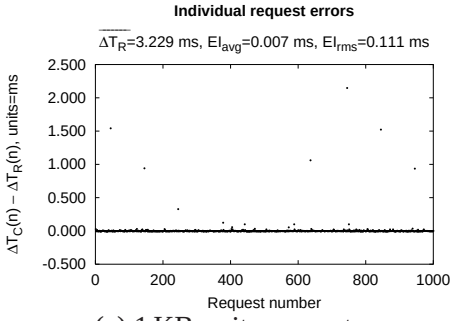
(b) 4 KB read requests



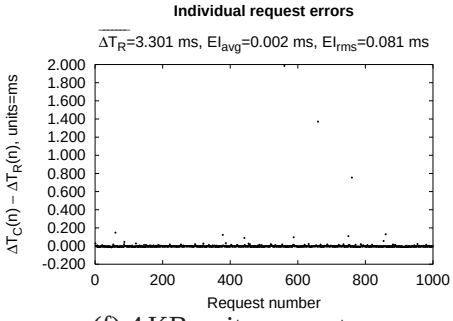
(c) 32 KB read requests



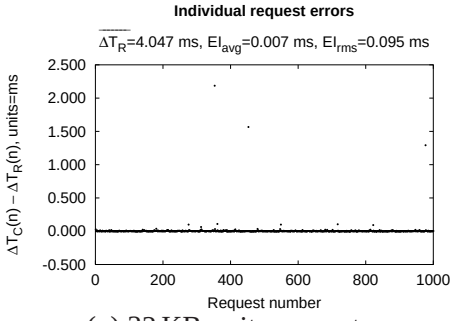
(d) 64 KB read requests



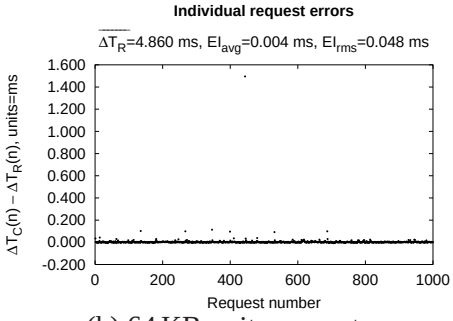
(e) 1 KB write requests



(f) 4 KB write requests

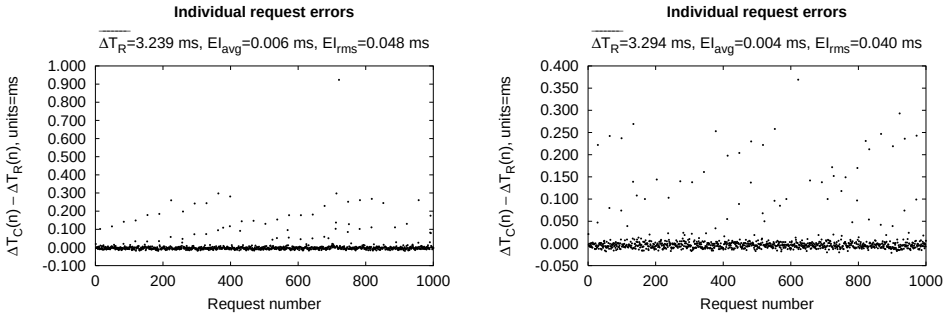


(g) 32 KB write requests



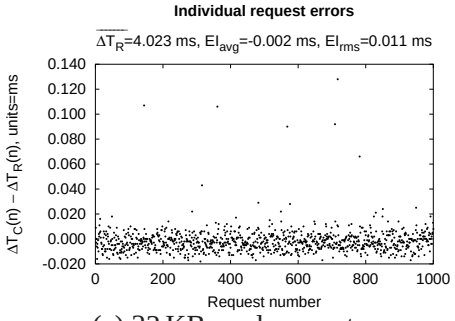
(h) 64 KB write requests

Figure D.9: $E_{1\rightarrow3}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=60** μ**s, Adaptec.**

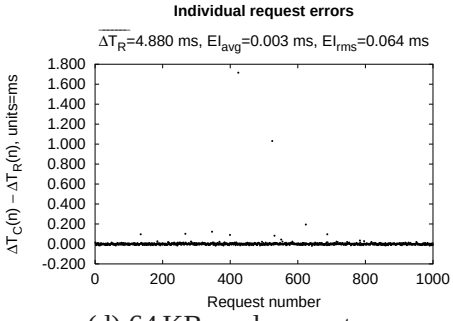


(a) 1 KB read requests

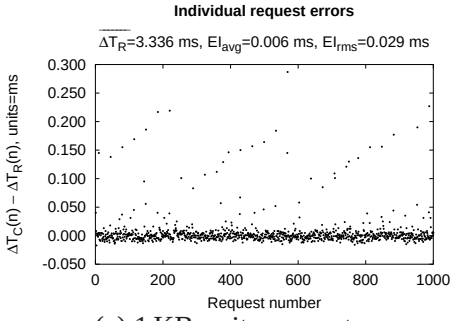
(b) 4 KB read requests



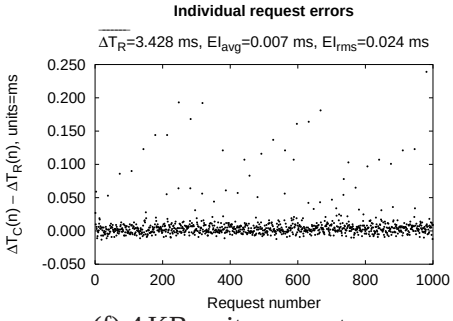
(c) 32 KB read requests



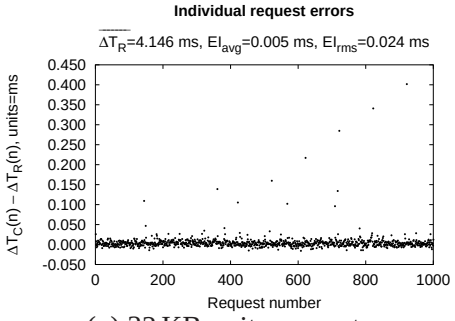
(d) 64 KB read requests



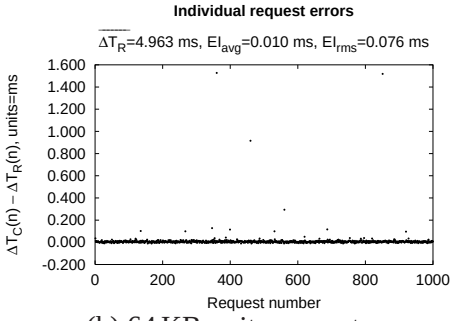
(e) 1 KB write requests



(f) 4 KB write requests

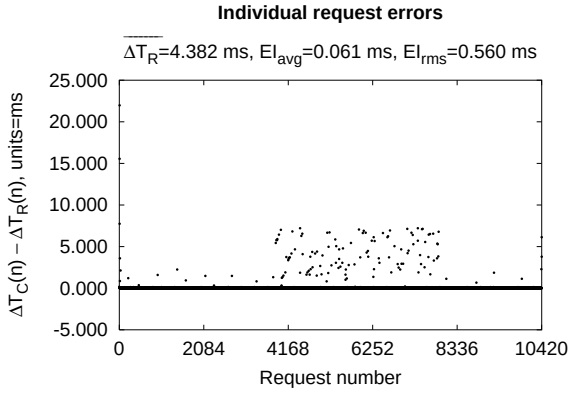


(g) 32 KB write requests

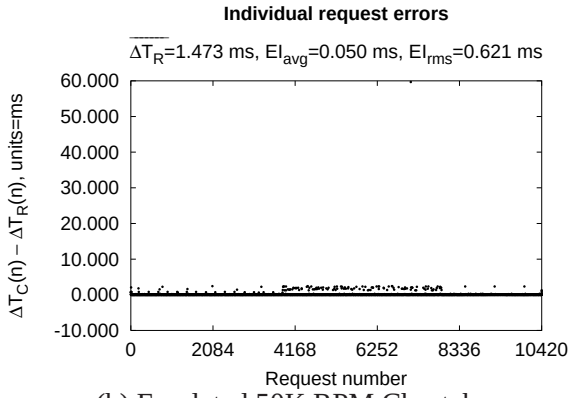


(h) 64 KB write requests

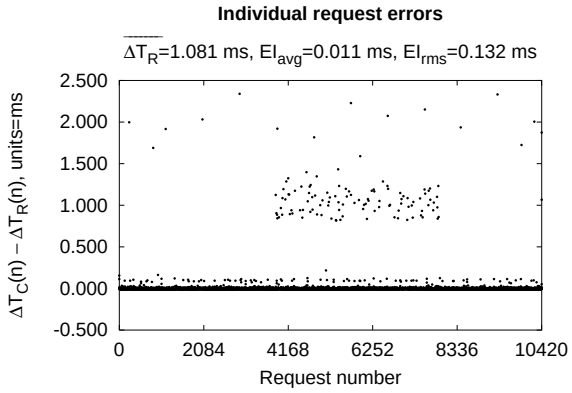
Figure D.10: $E_{1\rightarrow3}$**:** $\Delta T_{lookahead}$**=30** μ**s,** ΔT_{skew}**=151** μ**s, QLogic.**



(a) Emulated Cheetah

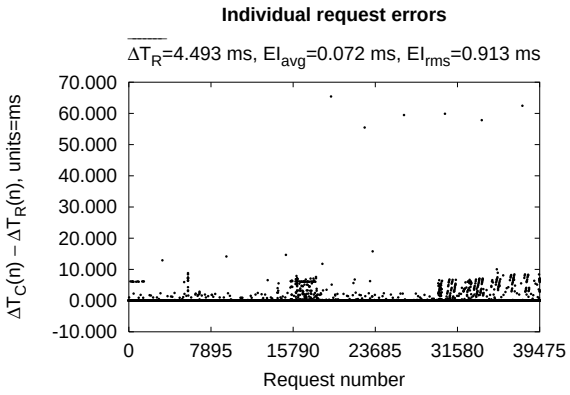


(b) Emulated 50K RPM Cheetah

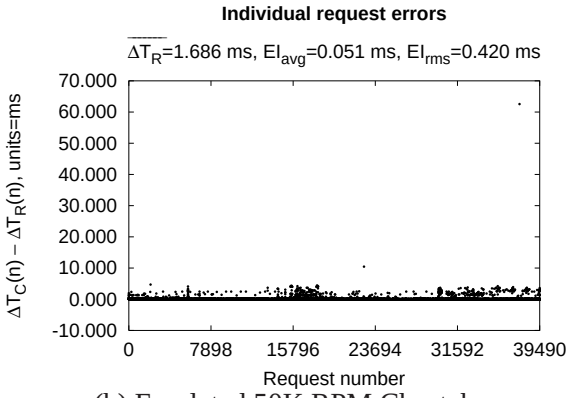


(c) Emulated MEMS-based storage device

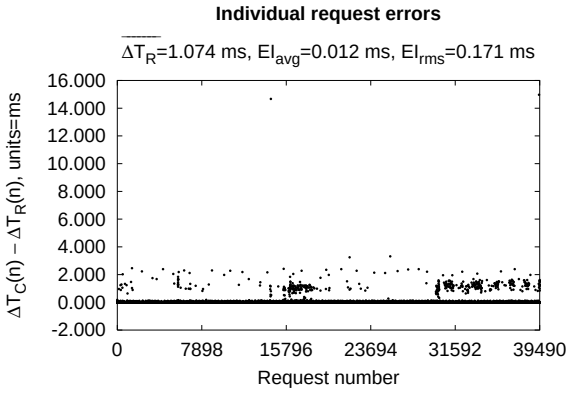
Figure D.11: PostMark: Emulated Cheetah, Emulated 50K RPM, Emulated MEMS.



(a) Emulated Cheetah

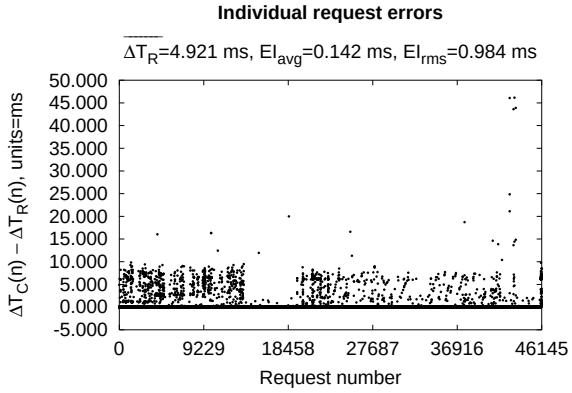


(b) Emulated 50K RPM Cheetah

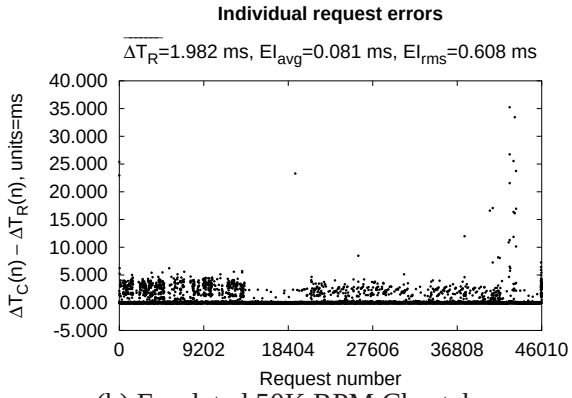


(c) Emulated MEMS-based storage device

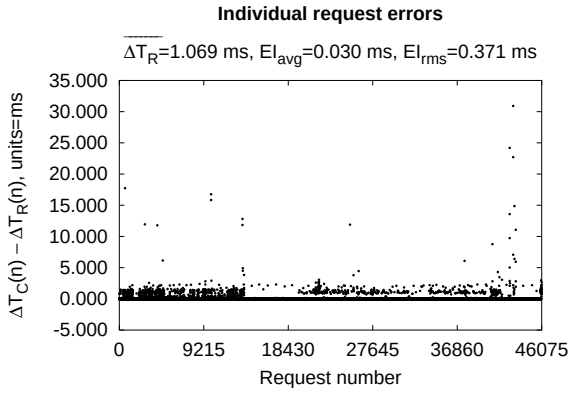
Figure D.12: SSH-build: Emulated Cheetah, Emulated 50K RPM, Emulated MEMS.



(a) Emulated Cheetah



(b) Emulated 50K RPM Cheetah



(c) Emulated MEMS-based storage device

Figure D.13: Linux-build: Emulated Cheetah, Emulated 50K RPM, Emulated MEMS.

BIBLIOGRAPHY

[1] 3Com. 3Com Embedded Firewall Architecture for E-Business. Technical Brief 100969-001. 3Com Corporation, April 2001. 117, 133

[2] 3Com. Administration Guide, Embedded Firewall Software. Documentation. 3Com Corporation, August 2001. 133

[3] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: programming model, algorithms and evaluation. *Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 3–7 October 1998), pages 81–91. ACM, 1998. 120, 137

[4] Marcos K. Aguilera, Minwen Ji, Mark Lillibridge, John MacCormick, Erwin Oertli, Dave Andersen, Mike Burrows Timothy Mann, and Chandramohan A. Thekkath. Block-level security for network-attached disks. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2003), pages 159–174. USENIX Association, 2003. 134

[5] 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. 24

[6] Guillermo A. Alvarez, Walter A. Burkhard, and Flaviu Cristian. Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering. *ACM International Symposium on Computer Architecture*, June 1997. 10

[7] Khalil S. Amiri. *Scalable and manageable storage systems*. PhD thesis, published as Technical report CMU–CS–00–178. Carnegie Mellon University, December 2000. v

[8] Eric Anderson. Simple table-based modeling of storage devices. SSP Technical Report HPL–SSP–2001–4. HP Laboratories, July 2001. 20

[9] 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. 38

[10] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. *IEEE Symposium on Security and Privacy* (Oakland, CA, 4–7 May 1997), pages 65–71. IEEE Computer Society Press, 1997. 134

[11] Ismail Ari, Ahmed Amer, Robert Gramacy, Ethan L. Miller, Scott A. Brandt, and Darrell D. E. Long. ACME: Adaptive Caching Using Multiple Experts. *Workshop on Distributed Data and Structures* (Paris, France, March 2002), 2002. 10

[12] Stefan Axelsson. Research in intrusion-detection systems: a survey. Technical report 98–17. Department of Computer Engineering, Chalmers University of Technology, December 1998. 120

[13] Rajive Bagrodia, Stephen Docy, and Andy Kahn. Parallel simulation of parallel file systems and I/O programs. *ACM International Conference on Supercomputing* (San Jose, CA, 15–21 November 1997), pages 1–17. ACM, 1997. 24

[14] Henry G. Baker. The Treadmill: real-time garbage collection without motion sickness. *SIGPLAN Notices*, **27**(3):66–70, March 1992. 152

[15] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. *ACM Symposium on Operating System Principles* (Asilomar, Pacific Grove, CA). Published as *Operating Systems Review*, **25**(5):198–212, 13–16 October 1991. 159

[16] Rakesh Barve, Elizabeth Shriver, Phillip B. Gibbons, Bruce K. Hillyer, Yossi Matias, and Jeffrey Scott Vitter. Modeling and optimizing I/O throughput of multiple disks on a bus. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Atlanta, GA, 1–4 May 1999). Published as *ACM SIGMETRICS Performance Evaluation Review*, **27**(1):83–92. ACM Press, 1999. 9

[17] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. *USENIX Annual Technical Conference* (New Orleans, LA, 16–20 January 1995), pages 277–288. USENIX Association, 1995. 159

[18] Patrick Bohrer, Mootaz Elnozahy, Ahmed Gheith, Charles Lefurgy, Tarun Nakra, James Peterson, Ram Rajamony, Ron Rockhold, Hazim Shafi, Rick Simpson, Evan Speight, Kartik Sudeep, Eric Van Hensbergen, and Lixin Zhang. Mambo—A full system simulator for the PowerPC architecture. *ACM SIGMETRICS Performance Evaluation Review*, **31**(4):8–12, March 2004. 11

[19] William J. Bolosky, Scott Corbin, David Goebel, and John R. Douceur. Single instance storage in Windows 2000. *USENIX Windows Systems Symposium* (Seattle, WA, 3–4 August 2000), pages 13–24. USENIX Association, 2000. 138

[20] Peter Bosch and Sape Mullender. PFS: A distributed and customizable file system. *Fifth International Workshop on Object-Orientation in Operating Systems* (Seattle, WA, 27–28 October 1996), pages 78–82. IEEE Computer Society, 1996. 24

[21] Daniel P. Bovet and Marco Cesati. *Understanding the Linux kernel*. O'Reilly & Associates, 2001. 159, 164

[22] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in network simulation. *Computer*, **33**(5):59–67. IEEE Computer Society, May 2000. 24

[23] Chappell Brown. Microprobes promise a new memory option. *Electronic Engineering Times*, **6**:41–44. CMP Media Inc., 12 January 1998. 143

[24] John S. Bucy and Gregory R. Ganger. The DiskSim simulation environment version 3.0 reference manual. Technical Report CMU–CS–03–102. Department of Computer Science Carnegie-Mellon University, Pittsburgh, PA, January 2003. 37, 42

[25] 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. 11

[26] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, **14**(4):311–343, November 1996. 138

[27] Rémy Card, Theodore Ts'o, and Stephen Tweedie. Design and implementation of the Second Extended Filesystem. *First Dutch International Symposium on Linux* (Amsterdam, The Netherlands, December 1994), 1994. 124

[28] 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. 143, 145, 148, 150, 154

[29] L. Richard Carley, Rany Tawfik El-Sayed, David F. Guillou, Fernando Alfaro, Gary K. Fedder, Steven W. Schlosser, John L. Griffin, David F. Nagle, Gregory R. Ganger, and James Bain. MEMS memory elements. *Non-volatile Memory Technology Symposium* (San Diego, CA, 07–08 November 2001), pages 1–5, 2001. 143

[30] Scott Carson and Sanjeev Setia. Optimal write batch size in log-structured file systems. *USENIX Workshop on File Systems* (Ann Arbor, MI, 21–22 May 1992), pages 79–91. Department of Computer Science, University of Maryland, 1992. 157, 158

[31] Edward Chang and Hector Garcia-Molina. Reducing initial latency in a multimedia storage system. *International Workshop on Multi-Media Database Management Systems* (Blue Mountain Lake, NY), pages 2–11, 14–16 August 1996. 157

[32] Edward Chang and Hector Garcia-Molina. Effective memory use in a media server. *International Conference on Very Large Databases* (Athens, Greece.), pages 496–505. Morgan Kaufmann Publishers, Inc., 26–29 August 1997. 157

[33] Fay W. Chang. *Using speculative execution to automatically hide I/O latency*. PhD thesis, published as CMU–CS–01–172. School of Computer Science, Carnegie Mellon University, 07 December 2001. v

[34] Chia Chao, Robert English, David Jacobson, Alexander Stepanov, and John Wilkes. Mime: high performance parallel storage device with strong recovery guarantees. Technical report HPL-92-9. 18 March 1992. 120

[35] 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. 22

[36] Peter M. Chen and Brian D. Noble. When virtual is better than real. *Hot Topics in Operating Systems* (Elmau, Germany, 20–22 May 2001), pages 133–138. IEEE Comput. Soc., 2001. 120

[37] Peter M. Chen and David A. Patterson. Maximizing performance in a striped disk array. *ACM International Symposium on Computer Architecture* (Seattle, WA), pages 322–331, 28–31 May 1990. 21

[38] IBM Corporation. Adaptive power management for mobile hard drives, 1999. 151, 154

[39] William V. Courtright II. *A transactional approach to redundant disk array implementation*. PhD thesis, published as Technical Report CMU–CS–97–141. School of Computer Science, Carnegie Mellon University, 15 May 1997. v, 22

[40] William V. Courtright II, Garth Gibson, M. Holland, and J. Zelenka. A structured approach to redundant disk array implementation. *IEEE International Computer Performance and Dependability Symposium* (Urbana-Champaign, IL, 04–06 September 1996), pages 11–20. IEEE, 1996. 10

[41] William V. Courtright II, Garth Gibson, Mark Holland, LeAnn Neal Reilly, and Jim Zelenka. RAIDframe: a rapid prototyping tool for RAID systems. CMU–CS–97–142. June 1997. 21

[42] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The Logical Disk: a new approach to improving file systems. *ACM Symposium on Operating System Principles* (Asheville, NC, 5–8 December 1993), pages 15–28, 1993. 163

[43] D. E. Denning, T. F. Lunt, R. R. Schell, M. Heckman, and W. Shockley. A multilevel relational data model. *IEEE Symposium on Security and Privacy* (Oakland, CA, 27–29 April 1987), pages 220–234. IEEE Computer Society Press, 1987. 120

[44] Dorothy E. Denning. *Information warfare and security*. Addison-Wesley, 1999. 118

[45] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Atlanta, GA, 1–4 May 1999). Published as *ACM SIGMETRICS Performance Evaluation Review*, **27**(1):59–70. ACM Press, 1999. 159

[46] Fred Douglis and Arun Iyengar. Application-specific delta-encoding via resemblance detection. *USENIX Annual Technical Conference* (San Antonio, TX, 09–14 June 2003), pages 113–126. USENIX Association, 2003. 138

[47] Fred Douglis, P. Krishnan, and Brian Bershad. Adaptive disk spin-down policies for mobile computers. *USENIX Symposium on Mobile and Location Independent Computing* (Ann Arbor, MI). Usenix Association, 10–11 April 1995. 10

[48] Fred Douglis, P. Krishnan, and Brian Marsh. Thwarting the power-hungry disk. *Winter USENIX Technical Conference* (San Francisco, CA), pages 292–306. USENIX Association, Berkeley, CA, 17–21 January 1994. 154

[49] Ann L. Drapeau and Randy H. Katz. Striping in large tape libraries. *Supercomputing '93* (Portland, OR. 15–19 November 1993), pages 378–387. IEEE Computing Society Press, November 1993. 21

[50] 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. 24

[51] Gary K. Fedder, Suresh Santhanam, Michael L. Reed, Steve C. Eagle, David F. Guillou, Mike S.-C. Lu, and L. Richard Carley. Laminated high-aspect-ratio microstructures in a conventional CMOS process. *IEEE Micro Electro Mechanical Systems Workshop* (San Diego, CA), pages 13–18, 11–15 February 1996. 144

[52] Stephanie Forrest, Setven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for UNIX processes. *IEEE Symposium on Security and Privacy* (Oakland, CA, 6–8 May 1996), pages 120–128. IEEE, 1996. 120

[53] Fujitsu Limited. *MAN3735MC/MP Series, MAN3367MC/MP Series, MAN3184MC/MP Series Disk Drives Product/Maintenance Manual*, Specification number C141-E128-01EN, Edition 01, June 2001. 44

[54] Eran Gabber, Jeff Fellin, Michael Flaster, Fengrui Gu, Bruce Hillyer, Wee Teck Ng, Banu Ozden, and Elizabeth Shriver. StarFish: highly-available block storage. *USENIX Annual Technical Conference* (San Antonio, TX, 09–14 June 2003), pages 151–164. USENIX Association, 2003. 22, 45

[55] Eran Gabber and Elizabeth Shriver. Lets put NetApp and CacheFlow out of business. *SIGOPS European Workshop* (Kolding, Denmark), pages 85–90, 17–20 Sept. 2000. 157, 159

[56] Greg Ganger and Jiri Schindler. Database of validated disk parameters for DiskSim. http://-www.pdl.cmu.edu/DiskSim/diskspecs.html. 21, 149

[57] Gregory 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. 12

[58] Gregory R. Ganger. Blurring the line between OSs and storage devices. Technical report CMU–CS–01–166. Carnegie Mellon University, December 2001. 137

[59] Gregory R. Ganger. *System-oriented evaluation of I/O subsystem performance*. PhD thesis, published as CSE–TR–243–95. University of Michigan, Ann Arbor, MI, June 1995. 2

[60] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: exploiting disk bandwidth for small files. *USENIX Annual Technical Conference* (Anaheim, CA), pages 1–17, January 1997. 157, 159, 167

[61] Gregory R. Ganger and David F. Nagle. Better security via smarter devices. *Hot Topics in Operating Systems* (Elmau, Germany, 20–22 May 2001), pages 100–105. IEEE, 2001. 120

[62] 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. 22

[63] 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. 2, 10, 11, 12

[64] 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. 21, 151

[65] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. *NDSS* (San Diego, CA, 06–07 February 2003). The Internet Society, 2003. 120

[66] Sanjay Ghemawat. *The modified object buffer: a storage management technique for object-oriented databases*. PhD thesis. Massachusetts Institute of Technology, Cambridge, MA, 7 September 1995. 157, 159

[67] Dominic Giampaolo. *Practical file system design with the Be file system*. Morgan Kaufmann, 1998. 159

[68] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. *Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 3–7 October 1998). Published as *SIGPLAN Notices*, **33**(11):92–103, November 1998. 22

[69] Howard Gobioff. *Security for a high performance commodity storage subsystem*. PhD thesis, published as Technical Report CMU–CS–99–160. School of Computer Science, Carnegie Mellon University, July 1999. v, 122, 134

[70] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. *Winter USENIX Technical Conference* (New Orleans, LA, 16–20 January 1995), pages 201–212. USENIX Association, 1995. 10

[71] Leana Golubchik, Richard R. Muntz, and Richard W. Watson. Analysis of striping techniques in robotic storage libraries. *IEEE Symposium on Mass Storage Systems* (Monterey, CA, 11–14 September 1995), pages 225–238. IEEE, 1995. 21

[72] John Linwood Griffin, Jiri Schindler, Steven W. Schlosser, John S. Bucy, and Gregory R. Ganger. Timing-accurate storage emulation. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 75–88. USENIX Association, 2002. 29, 40

[73] 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. 19, 21, 143, 145, 147, 151

[74] 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 Association, 2000. 143

[75] The Grugg. Defeating forensic analysis on Unix. *Phrack Magazine*, **11**(59):6, 28 July 2002. 121

[76] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mary Jane Irwin, N. Vijaykrishnan, and Mahmut Kandemir. Using complete machine simulation for software power estimation: the SoftWatt approach. *IEEE Symposium on High-Performance Computer Architecture* (Cambridge, MA, 2–6 February 2002), pages 141–150. IEEE Computer Society, 2002. 14

[77] Sudhanva Gurumurthi, Jianyong Zhang, Anand Sivasubramaniam, Mahmut Kandemir, Hubertus Franke, N. Vijaykrishnan, and Mary Jane Irwin. Interplay of energy and performance for disk arrays running transaction processing workloads. *IEEE International Symposium on Performance Analysis of Systems and Software* (Austin, TX, 6–8 March 2003), pages 123–132. IEEE, 2003. 10

[78] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 2nd ed.* Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1995. 147

[79] Stephen Alan Herrod. *Using complete machine simulation to understand computer system behavior*. PhD thesis, published as Technical Report CS–TR–98–1603. Department of Computer Science, Stanford University, February 1998. 11, 21, 47

[80] Bruce Hillyer, Rajeev Rastogi, and Avi Silberschatz. Scheduling and data replication to improve tape jukebox performance. *International Conference on Data Engineering* (Sydney, NSW, Australia, 23–26 March 1999), pages 532–541. IEEE Computer Society, 1999. 10

[81] Bruce K. Hillyer and Avi Silberschatz. Random I/O scheduling in online tertiary storage systems. *ACM SIGMOD International Conference on Management of Data* (Montreal, Quebec, Canada, 4–6 June 1996), pages 195–204. ACM, 1996. 10

[82] Bruce K. Hillyer and Avi Silberschatz. On the modeling and performance characteristics of a serpentine tape drive. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Philadelphia, PA, 23–26 May 1996), pages 170–179, May 1996. 21

[83] David Hitz. An NFS file server appliance. Technical report. Network Appliance, August 1993. 21

[84] David Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. *Winter USENIX Technical Conference* (San Francisco, CA, 17–21 January 1994), pages 235–246. USENIX Association, 1994. 163

[85] Mark Calvin Holland. *On-line data reconstruction in redundant disk arrays*. PhD thesis, published as Technical Report CMU–CS–94–164. Department of Electrical and Computer Engineering, Carnegie Mellon University, 1994. v

[86] Mark Horowitz, Thomas Indermaur, and Ricardo Gonzalez. Low-power digital design. *IEEE Symposium on Low Power Electronics* (San Diego, CA, 10–12 October 1994), pages 8–11. IEEE, 1994. 154

[87] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, **6**(1):51–81. ACM, February 1988. 151

[88] Windsor Hsu, Alan Jay Smith, and Honesty C. Young. Characteristics of production database workloads and the TPC benchmarks. *IBM Systems Journal*, **40**(3):781–802. IBM Almaden Research Center, 2001. 136

[89] Windsor W. Hsu and Alan Jay Smith. Characteristics of I/O traffic in personal computer and server workloads. *IBM Systems Journal*, **42**(2):347–372. IBM, 2003. 136

[90] Windsor W. Hsu and Alan Jay Smith. The performance impact of I/O optimizations and disk improvements. *IBM Journal of Research and Development*, **48**(2):255–289. IBM, March 2004. 10

[91] Kien A. Hua, Khanh Vu, and Ta-Hsiung Hu. Improving RAID performance using a multibuffer technique. *International Conference on Data Engineering* (Sydney, NSW, Australia, 23–26 March 1999), pages 79–86. IEEE Computer Society, 1999. 10

[92] IBM Corporation. *IBM family of microdrives*, http://www.storage.ibm.com/hardsoft/diskdrdl/-micro/datasheet.pdf, 2000. 153, 154

[93] IBM Corporation. *IBM Travelstar 32GH, 30GT, and 20GN 2.5-inch hard disk drives*, http://-www.storage.ibm.com/hardsoft/diskdrdl/travel/32ghdata.pdf, 2000. 153, 154

[94] Raj Jain. *The art of computer systems performance analysis*. John Wiley & Sons, 1991. 20, 37

[95] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, and Deborah A. Wallach. Server operating systems. *ACM SIGOPS. European workshop: Systems support for worldwide applications* (Connemara, Ireland, September 1996), pages 141–148. ACM, 1996. 167

[96] Jeffrey Katcher. PostMark: a new file system benchmark. Technical report TR3022. Network Appliance, October 1997. 18, 94, 151, 166

[97] Kimberly Keeton and Randy H. Katz. The evaluations of video layout strategies on a high-bandwidth file server. *4th International Workshop on Network and Operating System Support for Digital Audio and Video* (Lancaster, England, UK.), pages 228–229, 3–5 November 1993. 157

[98] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Record*, **27**(3):42–52, September 1998. 120

[99] Gene H. Kim and Eugene H. Spafford. The design and implementation of Tripwire: a file system integrity checker. *Conference on Computer and Communications Security* (Fairfax, VA, 2–4 November 1994), pages 18–29. ACM, 1994. 118, 120

[100] Gene H. Kim and Eugene H. Spafford. Experiences with Tripwire: using integrity checkers for intrusion detection. CSD-TR-94-012. 21 February 1994. 120

[101] Tracy Kimbrel, Andrew Tomkins, R. Hugo Patterson, Brian Bershad, Pei Cao, Edward W. Felten, Garth A. Gibson, Anna R. Karlin, and Kai Li. A trace-driven comparison of algorithms for parallel prefetching and caching. *Symposium on Operating Systems Design and Implementation* (Seattle, WA, 28–31 October 1996). Published as *Operating Systems Review*, **30**(special issue):19–34, 1996. 21

[102] Calvin Ko, Manfred Ruschitzka, and Karl Levitt. Execution monitoring of security-critical programs in distributed systems: a specification-based approach. *IEEE Symposium on Security and Privacy* (Oakland, CA, 04–07 May 1997), pages 175–187. IEEE, 1997. 120

[103] David Kotz. Disk-directed I/O for MIMD multiprocessors. *Symposium on Operating Systems Design and Implementation* (Monterey, CA), pages 61–74. USENIX Association, 14–17 November 1994. 10

[104] David Kotz. Tuning STARFISH. PCS–TR96–296. Department of Computer Science, Dartmouth College, NH, October 1996. 24

[105] 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. 21, 57

[106] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, John M. Tracey, and T. J. Watson. Redundancy elimination within large collections of files. *USENIX Annual Technical Conference* (Boston, MA, 27 June–02 July 2004), pages 59–72, 2004. 138

[107] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, **21**(7):558–565, 1978. 60

[108] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, **5**(8):874–879, August 1994. 154

[109] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, November 2000), pages 169–177. ACM, 2000. 122

[110] Yung-Hsiang Lu, Tajana Šimunić, and Giovanni De Micheli. Software controlled power management. *7th International Workshop on Hardware/Software Codesign* (Rome, Italy), pages 157–161. ACM Press, 3–5 May 1999. 154

[111] 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. 10, 29, 142, 159

[112] Tara M. Madhyastha and Katherine Pu Yang. Physical modeling of probe-based storage. *IEEE Symposium on Mass Storage Systems* (April 2001). IEEE, 2001. 21

[113] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, **35**(2):50–58. IEEE, February 2002. 11, 15

[114] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. *ACM Symposium on Operating System Principles* (Saint-Malo, France, 5–8 October 1997). Published as *Operating Systems Review*, **31**(5):238–252. ACM, 1997. 157, 159

[115] Ann Marie Grizzaffi Maynard. Validation of a full system simulator, 8 September 1999. Available at `http://www.cs.utexas.edu/users/cart/simOS/reports.htm`. 14, 40

[116] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–197, August 1984. 157, 164

[117] Rajeev Nagar. *Windows NT File System Internals: A Developer's Guide*. O'Reilly & Associates, 1997. 163

[118] Nils Nieuwejaar and David Kotz. The Galley parallel file system. *Parallel Computing*, **23**(4–5):447–476, 1 June 1997. 22

[119] 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. 24

[120] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. *ACM Symposium on Operating System Principles* (Orcas Island, WA). Published as *Operating Systems Review*, **19**(5):15–24, December 1985. 159

[121] Ashish Palekar, Narendran Ganapathym, Anshul Chadda, and Robert D. Russel. Design and implementation of a Linux SCSI target for storage area networks. *5th Annual Linux Showcase & Conference* (Oakland, CA, 05–10 November 2001), pages 99–106. USENIX Association, 2001. 20, 45

[122] David A. Patterson, Peter Chen, Garth Gibson, and Randy H. Katz. Introduction to redundant arrays of inexpensive disks (RAID). *IEEE Spring COMPCON* (San Francisco, CA), pages 112–117, March 1989. 147

[123] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO, 3–6 December 1995). Published as *Operating Systems Review*, **29**(5):79–95, 1995. 138

[124] Russel Hugo Patterson III. *Informed prefetching and caching*. PhD thesis, published as Technical Report CMU–CS–97–204. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, December 1997. v

[125] Adam G. Pennington, John D. Strunk, John Linwood Griffin, Craig A. N. Soules, Garth R. Goodson, and Gregory R. Ganger. Storage-based intrusion detection: watching storage activity for suspicious behavior. *USENIX Security Symposium* (Washington, DC, 06–08 August 2003), 2003. 119, 122, 132, 136

[126] Florentina I. Popovici, Andrea C. Arpaci Dusseau, and Remzi H. Arpaci Dusseau. Robust, portable I/O scheduling with the disk mimic. *USENIX Annual Technical Conference* (San Antonio, TX, 09–14 June 2003), pages 297–310. IEEE, 2003. 20

[127] Quantum Corporation. *Quantum Atlas 10K 9.1/18.2/36.4 GB Ultra 160/m SCSI Hard Disk Drive Product Manual*, Publication number 81-119313-05, August 6, 1999. 147, 151

[128] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 89–101. USENIX Association, 2002. 138

[129] Erik Riedel. *Active disks—Remote execution for network-attached storage*. PhD thesis, published as Technical Report CMU–CS–99–177. School of Computer Science, Carnegie Mellon University, November 1999. v, 137

[130] Erik Riedel. Personal communication, September 1, 2004. 11

[131] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia applications. *International Conference on Very Large Databases* (New York, NY, 24–27 August, 1998). Published as *Proceedings VLDB*, pages 62–73. Morgan Kaufmann Publishers Inc., 1998. 120

[132] 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. 11, 151

[133] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Symposium on Operating System Principles* (Pacific Grove, CA, 13–16 October 1991). Published as *Operating Systems Review*, **25**(5):1–15, 1991. 18

[134] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, **10**(1):26–52. ACM Press, February 1992. 157, 159

[135] Chris Ruemmler and John Wilkes. UNIX disk access patterns. *Winter USENIX Technical Conference* (San Diego, CA, 25–29 January 1993), pages 405–420, 1993. 12, 152

[136] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, **27**(3):17–28, March 1994. 21, 34, 55

[137] Brian Sawert. *The programmer's guide to SCSI*. Addison Wesley Longman Inc., February 1998. 45

[138] Joel Scambray, Stuart McClure, and George Kurtz. *Hacking exposed: network security secrets & solutions*. Osborne/McGraw-Hill, 2001. 118

[139] Jiri Schindler. *Matching application access patterns to storage device characteristics*. PhD thesis, published as Technical Report CMU–PDL–03–109. Electrical and Computer Engineering Department, Carnegie Mellon University, May 2004. v, 157

[140] Jiri Schindler, Anastassia Ailamaki, and Gregory R. Ganger. Lachesis: robust database storage management based on device-specific performance characteristics. *International Conference on Very Large Databases* (Berlin, Germany, 9–12 September 2003). Morgan Kaufmann Publishing, Inc., 2003. 157

[141] Jiri Schindler and Gregory R. Ganger. Automated disk drive characterization. Technical report CMU–CS–99–176. Carnegie-Mellon University, Pittsburgh, PA, December 1999. 19, 21, 151, 157

[142] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 259–274. USENIX Association, 2002. 136, 157

[143] Jiri Schindler, Steven W. Schlosser, Minglong Shao, Anastassia Ailamaki, and Gregory R. Ganger. Atropos: a disk array volume manager for orchestrated use of disks. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2004). USENIX Association, 2004. 157

[144] Steven W. Schlosser. *Using MEMS-based storage devices in computer systems*. PhD thesis, published as Technical Report CMU–PDL–04–104. Electrical and Computer Engineering Department, Carnegie Mellon University, May 2004. v, 10, 143

[145] Steven W. Schlosser and Gregory R. Ganger. MEMS-based storage devices and standard disk interfaces: a square peg in a round hole? *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2004), pages 87–100. USENIX Association, 2004. 143

[146] 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. 11, 14, 136, 143

[147] Steven W. Schlosser, Jiri Schindler, Anastassia Ailamaki, and Gregory R. Ganger. Exposing and exploiting internal parallelism in MEMS-based storage. Technical Report CMU–CS–03–125. Carnegie-Mellon University, Pittsburgh, PA, March 2003. 143

[148] Seagate Technology LLC. *Cheetah 36ES Family: ST336706LW/LC, ST318406LW/LC Product Manual, Volume 1*, Publication number 100141982, Rev. B, October 2001. 44

[149] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. *Winter USENIX Technical Conference* (Washington, DC, 22–26 January 1990), pages 313–323, 1990. 10

[150] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File system logging versus clustering: a performance comparison. *USENIX Annual Technical Conference* (New Orleans, LA, 16–20 January 1995), pages 249–264. Usenix Association, 1995. 158

[151] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. *USENIX Annual Technical Conference* (San Diego, CA, 18–23 June 2000), pages 71–84, 2000. 18, 94, 166

[152] Elizabeth Shriver, Eran Gabber, Lan Huang, and Christopher A. Stein. Storage management for web proxies. *USENIX Annual Technical Conference* (Boston, MA, 25–30 June 2001), pages 203–216, 2001. 167

[153] Elizabeth Shriver, Arif Merchant, and John Wilkes. An analytic behavior model for disk drives with readahead caches and request reordering. *International Conference on Measurement and Modeling of Computer Systems* (Madison, WI., 22-26 June 1998). Published as *Perform. Eval. Rev.*, **26**(1):182–191. ACM, June 1998. 9, 21

[154] Elizabeth A. M. Shriver, Bruce Hillyer, and Abraham Silberschatz. Performance analysis of storage systems. *Performance Evaluation*, pages 33–50. IEEE, 2000. 9, 21

[155] Tracy F. Sienknecht, Rich J. Friedrich, Joe J. Martinka, and Peter M. Friedenbach. The implications of distributed data in a commercial environment on the design of hierarchical storage management. *Performance Evaluation*, **20**(1–3):3–25, May 1994. 159

[156] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically smart disk systems. *Conference on File and Storage Technologies* (San Francisco, CA, 31–02 April 2003), pages 73–88. USENIX Association, 2003. 120, 124, 137, 138

[157] Keith A. Smith and Margo Seltzer. A comparison of FFS disk allocation policies. *USENIX.96* (San Diego, CA., 22–26 January 1996), pages 15–25. USENIX Assoc., 1996. 159

[158] Lance Spitzner. Honeytokens: The Other Honeypot. Security Focus, 21 July, 2003. http://www.securityfocus.com/infocus/1713. 121

[159] Iceberg 9200 disk array subsystem. Storage Technology Corporation, 2270 South 88th Street, Louisville, CO 80028-4358, 9 June 1995. 163

[160] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: protecting data in compromised systems. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 165–180. USENIX Association, 2000. 122

[161] Adam Sweeney. Scalability in the XFS file system. *USENIX Annual Technical Conference* (San Diego, CA, 22–26 January 1996), pages 1–14, 1996. 163

[162] Chandramohan A. Thekkath, John Wilkes, and Edward D. Lazowska. Techniques for file system simulation. *Software—Practice and Experience*, **24**(11):981–999. John Wiley and Sons Ltd, November 1994. 24

[163] Niki C. Thornock, Xia-Hong Tu, and J. Kelly Flanagan. A stochastic disk I/O simulation technique. *Winter Simulation Conference* (Atlanta, GA, 07–10 December 1997), pages 1079–1086. ACM, 1997. 20

[164] Andrew Tomkins. *Practical and theoretical issues in prefetching and caching*. PhD thesis, published as Technical Report CMU–CS–97–181. Carnegie Mellon University, October 1997. v

[165] Transaction Processing Performance Council. TPC Benchmark D (Decision Support) Standard Specification, 1998. 152

[166] Tripwire Open Souce 2.3.1, August 2002. http://ftp4.sf.net/sourceforge/tripwire/tripwire-2.3.1-2.tar.gz. 129

[167] Mustafa Uysal, Guillermo A. Alvarez, and Arif Merchant. A modular, analytical throughput model for modern disk arrays. *International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems* (Cincinnati, OH, 15–18 August 2001), pages 183–192. IEEE, 2001. 9

[168] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. *Symposium on*

Operating Systems Design and Implementation (Boston, MA, 9–11 December 2002), pages 271–284. USENIX Association, 2002. 24

[169] Elizabeth Varki, Arif Merchant, Jianzhang Xu, and Xiaozhou Qiu. Issues and challenges in the performance analysis of real disk arrays. *Transactions on Parallel and Distributed Systems*, **15**(6):559–574. IEEE, June 2004. 9, 19, 21

[170] Werner Vogels. File system usage in Windows NT 4.0. *ACM Symposium on Operating System Principles* (Kiawah Island Resort, Charleston, South Carolina, 12–15 December 1999). Published as *Operating System Review*, **33**(5):93–109. ACM, December 1999. 159

[171] 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. 23, 29, 57, 120, 137

[172] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. *Symposium on Operating Systems Design and Implementation* (Boston, MA, 9–11 December 2002), pages 255–270. USENIX Association, 2002. 24

[173] Rajiv Wickremesinghe, Jeffrey S. Chase, and Jeffrey S. Vitter. Distributed computing with load-managed active storage. *IEEE International Symposium on High-Performance Distributed Computing* (Edinburgh, UK, 23–26 July 2002), pages 13–23. IEEE Computer Society, 2002. 23, 137

[174] John Wilkes. The Pantheon storage-system simulator. HP Laboratories Technical Report HPL–SSP–95–14. HP Labs, May 1996. 21

[175] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO). Published as *Operating Systems Review*, **29**(5):96–108, 3–6 December 1995. 153

[176] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, **14**(1):108–136, February 1996. 10, 163

[177] Theodore M. Wong and John Wilkes. My cache or yours? Making storage more exclusive. *USENIX Annual Technical Conference* (Monterey, CA, 10–15 June 2002), pages 161–175, 2002. 10

[178] Theodore Ming-Tao Wong. *Decentralized recovery for survivable storage systems*. PhD thesis, published as Technical Report CMU–CS–04–119. School of Computer Science, Carnegie Mellon University, May 2004. v

[179] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling algorithms for modern disk drives. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Nashville, TN, 16–20 May 1994), pages 241–251. ACM Press, 1994. 148

[180] Xiang Yu, Benjamin Gum, Yuqun Chen, Randolph Y. Wang, Kai Li, Arvind Krishnamurthy, and Thomas 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. 9

[181] John Zedlewski, Sumeet Sobti, Nitin Garg, Fengzhou Zheng, Arvind Krishnamurthy, and Randolph Wang. Modeling hard-disk power consumption. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2003), pages 217–230. USENIX Association, 2003. 10

[182] Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ronald Perez, and Reiner Sailer. Secure Coprocessor-based Intrusion Detection. *ACM SIGOPS European Workshop* (Saint-Emilion, France, September 2002). ACM, 2002. 120, 122