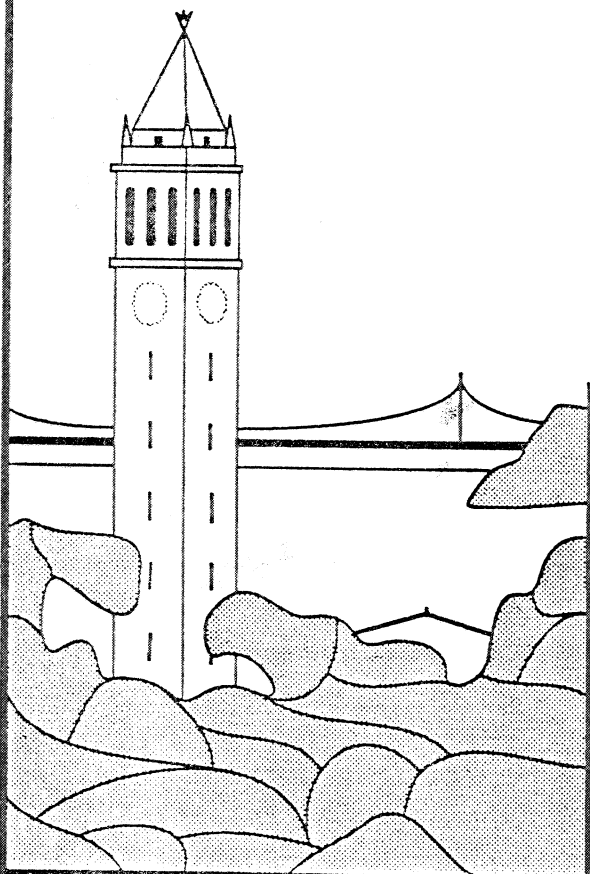


**Input/Output Performance Evaluation:  
Self-Scaling Benchmarks, Predicted Performance**

*Peter Ming-Chien Chen*



Report No. UCB/CSD-92-714

November 1992

Computer Science Division (EECS)  
University of California, Berkeley  
Berkeley, CA 94720

**Input-Output Performance Evaluation:  
Self-Scaling Benchmarks, Predicted Performance**

**by**

**Peter Ming-Chien Chen**

**B.S. (Pennsylvania State University) 1987  
M.S. (University of California at Berkeley) 1989**

**A dissertation submitted in partial satisfaction of the**

**requirements for the degree of**

**Doctor of Philosophy**

**in**

**Computer Science**

**in the**

**GRADUATE DIVISION**

**of the**

**UNIVERSITY of CALIFORNIA at BERKELEY**

**Committee in Charge:**

**Professor David A. Patterson, Chair  
Professor Randy H. Katz  
Professor Ronald Wolff**

The dissertation of Peter Ming-Chien Chen is approved:

D.A. Patterson 11/23/92  
Chair Date

Randy H. Katz 11/23/92  
Date

P.W. Wong 11/15/92  
Date

University of California at Berkeley

1992

**Input-Output Performance Evaluation:  
Self-Scaling Benchmarks, Predicted Performance**

**Copyright © 1992**

**by**

**Peter Ming-Chien Chen**



Abstract

# Input-Output Performance Evaluation: Self-Scaling Benchmarks, Predicted Performance

by

Peter Ming-Chien Chen

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor David A. Patterson, Chair

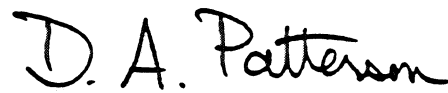
Over the past 20 years, processor performance has been growing much faster than input/output (I/O) performance. As this occurs, overall system speed becomes more and more limited by the speed of I/O systems and hence I/O systems are evolving to keep up with processor performance. This evolution renders current I/O performance evaluation techniques obsolete or irrelevant, despite their increasing importance. This dissertation investigates two new ideas in I/O evaluation, self-scaling benchmarks and predicted performance.

This dissertation's self-scaling benchmark seeks to measure and report relevant workloads for a wide range of input/output systems. To do so, it scales aspects of its workload to account for the differences in I/O systems. For example, it dynamically discovers the size of the system's file cache and reports how performance varies both in and out of the file cache. The

general approach taken is to scale based on the range of workloads the system performs well.

The self-scaling benchmark helps the evaluator gain insight into the system's performance by displaying how performance varies against each of five workload parameters: amount of file space, request size, fraction of reads, fraction of sequential accesses, and number of simultaneous accesses. The utility of the benchmark is demonstrated by running it on a wide variety of I/O systems, ranging from a single disk, low-end workstation to a mini-supercomputer with an array of four disks. On each system, the benchmark helps provide performance insights, such as the size of the file cache, the performance increases due to larger requests, the file cache's write policy, and the benefits of higher workload concurrency.

Predicted performance restores the ability to compare two machines on the same workload, which was lost in the self-scaling benchmark. Further, it extends this ability to workloads that have not been measured by estimating performance based on the graphs from the self-scaling benchmark. Prediction is accurate to within 10-15% over a wide range of I/O workloads and systems. This high level of accuracy demonstrates how a large workload space can be described using a few tens of points and a simple product form performance equation.

A handwritten signature in black ink that reads "D. A. Patterson". The signature is written in a cursive style with a horizontal line underneath it.

Professor David A. Patterson, Chair

*Dedicated to my wife, Janet*  
*for your help in persevering,*  
*your reminders for me to keep an eternal perspective,*  
*and your unconditional love and commitment.*

# Table of Contents

<b>Chapter 1: Introduction</b>	<b>1</b>
1.1 <b>Why I/O Performance?</b>	<b>1</b>
1.2 <b>Overview of the Dissertation</b>	<b>3</b>
1.3 <b>Metrics</b>	<b>4</b>
1.4 <b>Trends in I/O Systems</b>	<b>7</b>
1.5 <b>References</b>	<b>12</b>
<b>Chapter 2: Previous Work</b>	<b>17</b>
2.1 <b>I/O Benchmarks</b>	<b>17</b>
2.2 <b>The Ideal I/O Benchmark</b>	<b>18</b>
2.3 <b>System Platforms</b>	<b>20</b>
2.4 <b>Overview of Current I/O Benchmarks</b>	<b>21</b>
2.4.1 <b>Application Benchmarks</b>	<b>22</b>
2.4.1.1 <b>Andrew</b>	<b>22</b>
2.4.1.2 <b>TPC-B</b>	<b>23</b>
2.4.1.3 <b>Sdet</b>	<b>25</b>
2.4.2 <b>Synthetic Benchmarks</b>	<b>26</b>
2.4.2.1 <b>Bonnie</b>	<b>26</b>
2.4.2.2 <b>IOStone</b>	<b>27</b>
2.4.2.3 <b>Sample Scientific Workload</b>	<b>28</b>
2.4.2.4 <b>LADDIS</b>	<b>28</b>
2.5 <b>Critique of Current Benchmarks</b>	<b>30</b>
2.6 <b>References</b>	<b>32</b>
<b>Chapter 3: Workload Model</b>	<b>36</b>

3.1	Overview	36
3.2	I/O Workloads Used in Research	37
3.3	I/O Tracing Studies	38
3.4	Willy	40
3.4.1	Workload Parameters	40
3.4.1.1	Locality	40
3.4.1.2	Request Characteristics	41
3.4.1.3	Load	42
3.4.2	Other Workload Issues	42
3.5	Modeling Real Applications	43
3.6	Representativeness of Willy	44
3.7	References	46
<b>Chapter 4: A Self-Scaling Benchmark</b>		<b>49</b>
4.1	Overview	49
4.2	Single Parameter Graphs	50
4.3	The Knee Point	53
4.4	A Global Knee Self-Scaling Benchmark	58
4.4.1	Example Run of Global-Knee, Self-Scaling Benchmark	59
4.4.2	Problems	60
4.5	A Better Self-Scaling Benchmark	62
4.5.1	Examples	64
4.5.1.1	SPARCstation 1+	64
4.5.1.2	DECstation 5000/200	66
4.5.1.3	HP 730	69
4.5.1.4	Convex	70

4.5.1.5	Solbourne	70
4.5.1.6	Raw Disk Interface	74
4.5.1.7	Client-Server	74
4.5.1.8	Unannounced Workstation	75
4.5.2	Running Time	77
4.6	Conclusions	78
4.7	References	79
<b>Chapter 5: Predicted Performance</b>		<b>81</b>
5.1	Introduction	81
5.2	Prediction Models	82
5.3	Verification of Prediction Model	84
5.3.1	SPARCstation 1+	86
5.3.2	DECstation 5000/200, HP 730, Convex C240	88
5.4	Comparison Against Orthogonal Sampling	98
5.5	Application of Predicted Performance—Performance Ratios	100
5.6	Conclusions	100
5.7	References	103
<b>Chapter 6: Conclusions and Future Work</b>		<b>104</b>
6.1	Conclusions	104
6.2	Future Work	106
<b>Appendix A: Data Used in Prediction</b>		<b>109</b>
<b>Appendix B: Prediction Algorithm</b>		<b>120</b>
<b>Appendix C: Proof of Performance Equation</b>		<b>123</b>

# List of Figures

Figure 1.1	Contrasting trends of CPU and disk performance improvements	2
Figure 1.2	Example response time versus throughput graph	5
Figure 1.3	Combining multiple smaller disks to improve performance	9
Figure 2.1	TPC-B results	23
Figure 2.2	Sdet results	25
Figure 2.3	Current state of I/O benchmarks	30
Figure 4.1	Results from a self-scaling benchmark that scales all parameters	52
Figure 4.2	Workloads reported by a set of single parameter graphs	53
Figure 4.3	Difficulty in defining the knee	55
Figure 4.4	Global knee point with two parameters	57
Figure 4.5	Instability in graphs on the border of performance regions	61
Figure 4.6	Slope of uniqueBytes curve for SPARCstation 1+	63
Figure 4.7	Results from a better self-scaling benchmark for a SPARCstation 1+	65
Figure 4.8	Self-scaling benchmark for DECstation 5000/200—Part 1	67
Figure 4.9	Self-scaling benchmark for DECstation 5000/200—Part 2	68
Figure 4.10	Self-scaling benchmark for HP 730	69
Figure 4.11	Self-scaling benchmark for Convex C240	71
Figure 4.12	Self-scaling benchmark for Solbourne	72
Figure 4.13	Self-Scaling Results using the Raw Disk Interface	73
Figure 4.14	Self-Scaling Results for Two Client-Server Configurations	75
Figure 4.15	Self-scaling benchmark for unannounced workstation (beta release)	76
Figure 5.1	Predicting performance of unmeasured workloads	83
Figure 5.2	Predicting performance of four traditional benchmarks	85
Figure 5.3	Evaluation of prediction accuracy for SPARCstation 1+ with one disk	87

Figure 5.4	What parameters cause errors for SPARCstation 1+	89
Figure 5.5	Evaluation of prediction accuracy for Sprite DECstation 5000/200	90
Figure 5.6	What parameters cause errors for DECstation 5000/200	91
Figure 5.7	Evaluation of prediction accuracy for HP 730	92
Figure 5.8	What parameters cause errors for HP 730	93
Figure 5.9	Evaluation of prediction accuracy for Convex C240	94
Figure 5.10	What parameters cause errors for Convex C240	95
Figure 5.11	Enhanced prediction accuracy	97
Figure 5.12	Prediction accuracy using interpolation on an orthogonal sample	99
Figure 5.13	Measured versus predicted ratio	102



# List of Tables

Table 1.1	Metrics for two I/O systems	7
Table 1.2	Magnetic disk performance improvement over the past 20 years	8
Table 1.3	1992 storage capacity	10
Table 2.1	System platforms	20
Table 2.2	List of contacts for various benchmarks	21
Table 2.3	Results from the Andrew benchmark	22
Table 2.4	Results from Bonnie	26
Table 2.5	Results from IOStone	27
Table 2.6	Results from two sample scientific workloads	28
Table 3.1	Workload characterization of benchmarks/applications	44
Table 3.2	Representativeness of Willy	45
Table 4.1	Description of other machines used in self-scaling benchmark examples	64
Table 5.1	Summary of median prediction errors	96
Table 5.2	Workloads to be run on all systems	101
Table A.1	Raw data for SPARCstation 1+ scatter plot	112
Table A.2	Raw data for DECstation 5000/200 scatter plot	114
Table A.3	Raw data for HP 730 scatter plot	116
Table A.4	Raw data for Convex C240 scatter plot	119
Table A.5	Raw data for ratio prediction	119

# Acknowledgements

I am grateful to many people for helping me finish this thesis. My advisor, David Patterson, has been the best advisor I could imagine. Since the time he welcomed me on board the RAID project, he has been providing interesting research ideas, encouragement to persevere through graduate school, wise advice through the "fallacies and pitfalls" of an academic career, enthusiasm to learn and teach, and fatherly friendship. He has been a great role model for me as I begin my vocation as a university professor, and I hope to be, like him, a teacher and researcher who genuinely cares for my students.

Randy Katz has been a very supportive secondary advisor. As Principal Investigator of the Berkeley RAID project, he gave me the opportunity to be involved in an interesting research project, RAID-II. Through this project, I have gained valuable, hands-on implementation experience. I am thankful to Randy and my third reader, Ronald Wolff, for giving me insightful, timely feedback and suggestions for this thesis.

Most Ph.D. students have only one advisor, I was lucky enough to have not only Dave Patterson and Randy Katz but also another mentor, fellow graduate student (now professor) Garth Gibson. He, with Dave and Randy, guided me through graduate school and shaped the way I approach research problems and write papers.

I have enjoyed working with the RAID and Sprite project members. Students from both projects have been good sounding boards for this research. In particular, Garth Gibson and Ed Lee have helped shape much of my research, from discussions on striping units to I/O workloads. Mendel Rosenblum and John Hartman helped me understand file systems and file caches. Ken Shirriff helped me develop an informal proof about the global knee of Chapter 4. Ken Lutz taught me countless tips on how to debug real hardware in the context of RAID-II.

My parents have consistently supported me through life, teaching me the value of education and helping me form a sensible priority system. Without my parents, I would not have striven for an academic position and may not have persevered through graduate school. The friends I met through Campus Crusade for Christ have been my family in California and have given zest to my life here.

One of those friends in Campus Crusade for Christ eventually has become my closest family. My wife, Janet, has been a wonderful partner in life while I've written this dissertation. Our first one and a half years of marriage have been the best period of my life, both personally and academically.

Finally, I would like to explain, in story form, a truth I have understood while in graduate school. Let us say you are a judge and I am a drunk driver. Not only are you a judge, you are also my father. While driving through town, I crash through a very expensive store window. The penalty for this crime is either paying for the store window (worth, say, \$100,000) or working without pay for the store until the debt is paid off. As my father, you know I am broke and can not pay the \$100,000. The dilemma is this: naturally, you love me and do not want me to work the rest of my life paying back the store. But, being a just judge, you can not arbitrarily overlook my crime.

Here is a possible solution to this quandary: you could offer to pay the penalty for me, thus accepting the cost of my error on yourself. I need only accept your offer.

This hypothetical situation describes fairly closely our situation before God. All of us (including myself) are naturally selfish and have done wrong things, whether of the more heinous variety, such as murder, or the more everyday variety, such as hatred. The penalty for these wrongdoings is spiritual death, which is eternal separation from God. God, as a Father who loves us, does not wish for us to pay this penalty, but, nonetheless, the penalty must be paid for God to be just. He solves this dilemma by penalizing Himself in the form of Jesus.

Jesus's death was God's payment for our mistakes. God offers to apply this payment for each person, but not everyone has accepted His offer, either due to misunderstanding or due to a desire to trust solely in one's own ability to pay. If you are willing to accept God's payment for your mistakes, simply tell God of your acceptance and begin to yield to Him as your new master.

*For the wages of sin is death, but the free gift of God is eternal life in Christ Jesus our Lord.*

*(Romans 6:23)*

---

# Chapter 1

## Introduction

---

### 1.1. Why I/O Performance?

In the last decade, innovations in technology have led to extraordinary advances in computer processing speed. These advances have led many of those who evaluate a computer's performance to focus their attention on measuring processor performance to the near exclusion of all other metrics; some have even equated a computer system's performance with how well its CPU performs. This viewpoint, which makes system-wide performance synonymous with CPU speed, is becoming less and less valid. One way to demonstrate this declining validity is illustrated in Figure 1.1, where IBM disk performances represented by the throughput of accessing a random 8 KB block of data, and IBM mainframe CPU performance [Patterson90]. For the sake of comparison, both CPU and disk performance are normalized to their 1971 levels. As can

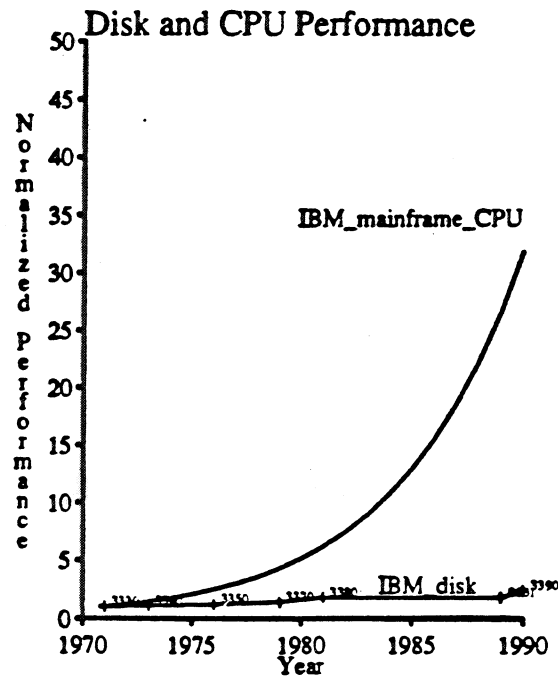


Figure 1.1: Contrasting trends of CPU and disk performance improvements. Over the past two decades, improvements in CPU performance have far outstripped those in disk performance. In this graph, CPU performance refers to IBM mainframe performance; disk performance refers to IBM disk (33x0 series) throughput on a random 8 KB access. Both are normalized to their 1971 levels. The data for IBM mainframe performance comes from Figure 1.1 on page 4 of [Patterson90].

readily be seen, over the past two decades, IBM mainframe CPU performance has increased more than 30-fold, while IBM disk performance has barely doubled. Microprocessor performance has increased even faster than mainframe performance [Myers86, Gelsinger89]. If CPU performance continues to improve at its current pace and disk performance continues to obtain more moderate improvements, eventually the performance of all applications that do any input or output (I/O) will be limited by that I/O component—further CPU performance improvements will be wasted [Amdahl67].

In light of this developing trend toward I/O-limited applications, I/O performance and architecture become increasingly more crucial to a system's overall performance. Researchers are rapidly developing new types of I/O architectures to match the improvements in processor performance; I contend that these new I/O architectures must be evaluated using new benchmarking techniques. In this dissertation, I propose a new approach to I/O benchmarks, which I've called *self-scaling evaluation and predicted performance*, that scales more comprehensively, provides more understanding of the system being measured, and estimates performance to within 10-15% over a wide range of I/O workloads and systems.

## 1.2. Overview of the Dissertation

The rest of this chapter presents background material on I/O performance evaluation. I discuss common metrics that researchers use to evaluate I/O systems. After this, I highlight current trends in I/O systems and how these trends affect I/O performance evaluation.

In Chapter 2, I survey and evaluate current I/O benchmarks, testing them on three Unix workstations. After doing so, I list desirable characteristics for I/O benchmarks and use these to critique current I/O benchmarks. I also summarize other research done in the modeling of I/O and file system performance.

In Chapter 3, I describe in detail the workload model that provides a framework for my new benchmarking approach. I discuss the process of developing the workload model and some alternatives. I then trace real applications and describe their I/O workload.

In Chapter 4, I propose the first part of my new approach to I/O evaluation—a *self-scaling benchmark*. This benchmark scales the workload used to evaluate a system based on that system's capabilities. I show that benchmarks must scale to be useful and that this method scales more effectively than any current benchmarking method. I also show how the self-scaling benchmark provides insight into a computer system by giving information on what

workload might be appropriate to run on each system. I demonstrate the utility of the self-scaling benchmark by showing benchmark results gathered on a wide variety of I/O systems, ranging from a one disk, low-end workstation to a four disk mini-supercomputer.

In Chapter 5, I propose the second part of my new approach—*predicted performance*, which uses measurements from a small set of workloads to accurately estimate performance for arbitrary workloads. I show that predicted performance estimates performance very accurately, within 10-15%, on a wide range of I/O systems and investigate how error is correlated to each workload parameter. I compare my method of predicting performance against using an orthogonal sampling of many workloads and show how my method gives lower error while using fewer workload measurements. I end by applying predicted performance to predict the relative performance ratio between two systems.

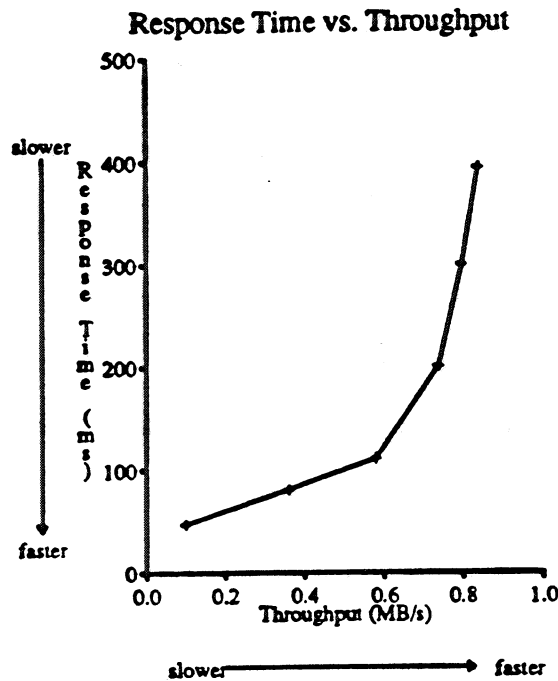
In Chapter 6, I conclude with a summary of the contributions made by my thesis and I look at some future directions for this research.

### 1.3. Metrics

More than other areas of computer performance evaluation, I/O evaluation involves a great variety of metrics. This section contains an overview of some of the metrics commonly used today in choosing and evaluating I/O systems. The value of most metrics depend strongly on the workload used.

The most basic metric for I/O performance is *throughput*. Throughput is a measure of speed—the rate at which an I/O system delivers data, and it is measured in two ways: I/O rate, measured in *accesses/second*, and data rate, measured in *bytes/second* or *megabytes/second (MB/s)*. I/O rate is commonly used for applications where the size of each request is small, such as transaction processing [Anon85]; data rate is commonly used for applications where the size of each request is large, such as scientific applications [Miller91a].





**Figure 1.2: Example response time versus throughput graph.** Increasing the utilization of a system usually leads to higher throughput but slower response time. This figure was adapted from [Chen90b].

---

*Response time* is the second basic performance metric for I/O systems, and it measures how long an I/O system takes to access data. This time can be measured in several ways. For example, one could measure I/O response time from the user's perspective, the operating system's perspective, or the disk controller's perspective, depending on what is considered the I/O system.

The usefulness of a I/O system not only includes *how fast* data can be accessed but also *how much* data can be stored. *Capacity* is not normally applied as a metric to non-storage components of a computer system, but it is an integral part of evaluating an I/O system. If capacity were ignored as a metric, tape and disk manufacturers would soon find their customers switching to solid-state (memory-based) storage systems, which offer much higher performance but

less capacity per dollar.

Because users store valuable data on I/O systems, they demand a level of reliability much higher than for other parts of a computer. If a memory chip develops a parity error, for example, the system will (hopefully) crash and be restarted. On the other hand, if a storage device develops a parity error in a database of bank accounts, banks could unwittingly lose millions of dollars. Thus, *reliability* is a metric uniquely important to storage systems.

*Cost*, of course, applies to all components in computer systems, particularly to disk subsystems, which are often the most expensive component in a large computer installation [Bodega89]. Cost is usually expressed as a composite metric, such as cost per capacity (dollars per MB) or throughput cost (dollars per MB/s).

Table 1.1, adapted from [Gibson91], shows the values of the above metrics for two different disk systems.

These five metrics—throughput, response time, capacity, reliability, and cost—are commonly used in various combinations to evaluate I/O systems. One popular combination is a response time versus throughput graph (Figure 1.2). These graphs vary a parameter, such as the number of users on the system, to display the tradeoff between improving throughput and degrading response time. With more users, the system can often be utilized more efficiently, which increases throughput. On the other hand, higher utilization leads to slower response times. Because a single performance number is easier to use than a full graph, many evaluators combine throughput and response time by reporting throughput at a given response time [Anon85, Chen90b]. The TPC-B benchmark, for example, reports maximum throughput with 90% of all requests completed within 2 seconds [TPCB90] (see Section 2.4.1.2).

Another composite metric is *data temperature*, which is defined as I/O rate divided by capacity [Katz90]. Data temperature measures how many I/Os per second a I/O system can support for a fixed amount of storage. This is a valuable metric for users who are limited by I/O

rate rather than capacity because it tells them that they should buy systems with high data temperature.

A general parameterizable composite metric can be formulated for any combination of the above metrics. For example, one could imagine a system administrator who wanted a system with the highest capacity per dollar, as long as it satisfied minimum demands for reliability, throughput, and response time.

#### 1.4. Trends in I/O Systems

To understand how developments in I/O systems are straining the capabilities of current I/O benchmarks, I highlight some of the trends in I/O systems in this section. I discuss advances in magnetic disk technology, arrays of disks, file caching and solid state disks, magnetic tape, and log-structured file systems.

Magnetic disks have long been the mainstay of I/O systems, but since 1970, disk performance has improved only modestly. Table 1.2 compares two disks, the IBM 3330, introduced in 1971, and the IBM 0661, introduced in 1989. The average yearly improvement in performance has inched forward at a few percent a year. Cost per capacity, on the other hand, has improved at a much faster pace, averaging a 23% reduction per year from 1977 to 1986 [Gibson91]. Moreover, individual disks have also been gradually decreasing in physical size and

Type of Metric	Specific Measure	IBM 3390	Redundant Disk Array of IBM 0661 disks
Throughput	Max Read I/O Rate	609 I/O's per second	3889 I/O's per second
Throughput	Max Read Data Rate	15 MB per second	130 MB per second
Response Time	Min Response Time	20 ms	20 ms
Capacity	GB	23 GB	22 GB
Reliability	Mean Time to Data Loss	6-28 years	753 years
Cost	\$ (estimated)	\$156,000 - \$260,000	\$67,000 - ?

Table 1.1: Metrics for two I/O systems. Above are the differences in the values of several types of metrics for an IBM 3390 disk system and a redundant disk array made of IBM 0661 3.5" drives [IBM0661]. This table is adapted from [Gibson91].

cost. The most common diameter of a disk in the 1970's and 1980's was 14". Those disks are disappearing and are being replaced with 5.25" and 3.5" diameter disks. The performance of these smaller disks is comparable to that of their larger, more expensive predecessors.

The trend toward smaller, less expensive disks makes it possible to combine many of them into a parallel I/O system known as a *disk array*. Arrays of multiple disks have been used for many years for special purposes [Johnson84] but they are only now becoming popular for general use. The list of companies developing or marketing disk arrays is quite long, and it includes Array Technology, Auspex, Ciprico, Compaq, Cray, Datamax, Hewlett-Packard, IBM, Imprimis, Intel Scientific, Intellistor, Maximum Strategy, Pacstor, SF2, Storage Concepts, Storage Technology, and Thinking Machines. Some analysts have projected that the disk array market will expand to \$8 billion by 1994 [Montgomery91].

The idea behind disk arrays is straightforward—combine many small disks and distribute data among them (Figure 1.3), which increases the aggregate throughput available to an application. An array of disks can service either many small accesses in parallel or cooperate to deliver a higher data rate to a single, large access [Patterson88, Gibson91, Livny87, Salem86]. Disk arrays compensate for the lower reliability inherent in using more disks by storing redundant, error-correcting information. Current research into disk arrays is focusing on 1) how to distribute (stripe) data across disks to get optimal performance [Chen90a, Lee91a, Lee91b, Weikum90] and 2) how to spread redundant information across disks to increase reliability and

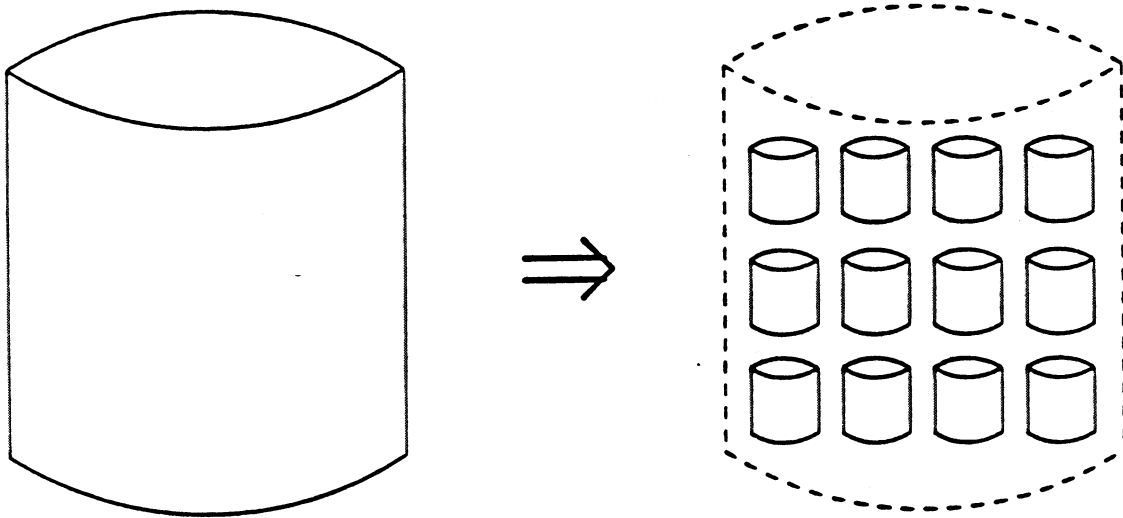
Metric	IBM 3330	IBM 0661	Average Yearly Improvement
Year Introduced	1981	1989	
Average Seek Time	30 ms	12.5 ms	5%
Average Rotational Delay	8.3 ms	7 ms	1%
Transfer Rate	806 KB/s	1700 KB/s	4%

**Table 1.2: Magnetic disk performance improvement over the past 20 years.** This table shows the slow average improvement in disk performance over the past 20 years. The IBM 3330 was introduced in 1981 and is 14 inches in diameter; the IBM 0661 was introduced in 1989 and is 3.5 inches in diameter [Harker81, IBM0661].

minimize the effects of disk failures [Holland92, Gibson91, Muntz90].

Although disk arrays improve throughput by using more disks to service requests, requests that are serviced by a single disk still see the same response time. Techniques for improving response time include file caches, disk caches, and solid state disks, all of which use dynamic RAM (random access memory). Caches can be located in a variety of places in a system's memory hierarchy [Smith85]. Two common places are the disk controller, as in the IBM 3990 disk cache [Menon87], and main memory, as in the Sprite operating system's file cache [Ousterhout88, Nelson88]. Response times for writes are decreased by writing the data to

---



**Figure 1.3: Combining multiple smaller disks to improve performance.** Performance of single disks is not improving rapidly (Table 1.2); however, disks are rapidly becoming physically smaller and cheaper. Disk arrays take advantage of this downsizing to provide higher aggregate throughput by simultaneously operating many small disks.

---

RAM, acknowledging the request, then transferring the data to disk asynchronously. However, this technique, called *write-behind*, leaves the data in RAM more vulnerable to system failures until it is written to disk. Some systems, such as the IBM 3990, mitigate this problem of vulnerability by storing the cached data in non-volatile memory, which is immune to power failures [Menon87]. As with any cache, read response time is decreased if the requested data is found in cache RAM.

Solid state disks are similar to caches in that they improve response time by storing requests in RAM rather than on magnetic disks. The principal difference between solid state disks and caches is that solid state disks speed up all accesses while caches speed up access only to the most commonly requested data. Although solid state disks costs 50-100 times more per capacity than magnetic disks [Gibson91], they are dramatically faster. Response times for solid state disks are commonly less than 3 ms [Cassidy89, Jones89], while response times for magnetic disks are approximately 10-30 ms.

Two I/O metrics have been addressed, throughput and response time. Dramatic improvements to capacity per cost have occurred in magnetic tapes (Table 1.3). These improvements are due in part to ever-increasing bit densities on tapes and partly to the acceptance of helical scan technology. Helical scan is a method of writing and reading tapes which can increase the capacity of a single tape from 0.1-0.2 GB to 5-20 GB [Katz91, Tan89, Vermeulen89]. Tapes

Device	Total Capacity	Cost/Capacity	Media Capacity	Latency
Magnetic Disk	1 GB	\$2,500/GB	1 GB	0.01 sec.
Dilog DAT Stacker	10 GB	\$527/GB	1.3 GB	75 sec.
Exabyte 120 Tape Library	500 GB	\$80/GB	5 GB	100 sec.
Metrum RSS-600 Tape Library	8700 GB	\$62/GB	14.5 GB	50 sec.

Table 1.3: 1992 storage capacity. This table, adapted from [Fine92], shows the extraordinary capacity of today's tape systems.

are extremely slow, however, with response times ranging from 20 seconds to a few minutes; throughput for these devices is less dismaying, ranging from 0.1 to 2.0 MB/s. Current research related to tape devices addresses how to migrate data from tape to faster storage [Smith81, Thanhardt88, Hac89, Miller91b, Henderson89], how to increase tape throughput using striping [Katz91], and how to decrease response times by prefetching and caching [Gibson92, Fine92].

Reported disk reliability has improved dramatically over the past ten years, though actual reliability has improved more slowly. The most common metric used to gauge reliability, *mean-time-to-failure*, has increased from 30,000 hours to 150,000-200,000 hours. This jump in apparent reliability comes mostly from changing the method by which mean-time-to-failure is computed and is not expected to continue improving as quickly [Gibson91].

I/O innovation is also taking place in file systems. A good example of how file systems have improved I/O system performance is the Log-Structured File System (LFS) [Rosenblum91, Ousterhout89], which allocates data on disk in the same order that it is written. This leads to highly sequentialized disk writes and so improves the sustainable disk write throughput.

Although the raw performance of I/O technology has improved much more slowly than processor technology, innovation such as file caches, disk arrays, robot-driven tape systems, and new file systems have helped close the gap. At the same time, these innovations have created new challenges for I/O benchmarks. For instance, solid state disks, file caches, and disk caches all use dynamic RAM (DRAM), whose capacity has been quadrupling every three years [Myers86]. Due to this rapid growth, benchmarks that had, at one time, exercised the disk system will no longer do so.

## 1.5. References

- [Amdahl67] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", *Proceedings AFIPS 1967 Spring Joint Computer Conference 30* (April 1967), 483-485.
- [Anon85] Anon and *et al.*, "A Measure of Transaction Processing Power", *Datamation*, 31, 7 (April 1, 1985), 112-118.
- [Bodega89] *National Science Foundation Workshop on Next Generation Secondary Storage Architecture*, National Science Foundation, Bodega Bay, CA, May 1989.
- [Cassidy89] C. Cassidy, "DEC's ESE20 Boosts Performance", *DEC Professional*, May 1989, 102-110.
- [Chen90a] P. M. Chen and D. A. Patterson, "Maximizing Performance in a Striped Disk Array", *Proceedings of the 1990 ACM SIGARCH Conference on Computer Architecture*, Seattle WA, May 1990, 322-331.
- [Chen90b] P. M. Chen, G. Gibson, R. H. Katz and D. A. Patterson, "An Evaluation of Redundant Arrays of Disks Using an Amdahl 5890", *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Boulder CO, May 1990.
- [Fine92] J. A. Fine, T. E. Anderson, M. D. Dahlin, J. Frew, M. Olson and D. A. Patterson, "Abstracts: A Latency-Hiding Technique for High-Capacity Mass Storage Systems", *Sequoia Technical Report 92/11*, University of California at Berkeley, March 1992.
- [Gelsinger89] P. P. Gelsinger, P. A. Gargini, G. H. Parker and A. Y. C. Yu, "Microprocessors Circa 2000", *IEEE Spectrum*, October 1989, 43-47.
- [Gibson91] G. A. Gibson, "Redundant Disk Arrays: Reliable, Parallel Secondary Storage", UCB/Computer Science Dpt. 91/613, University of California at Berkeley, December 1991. also available from MIT Press, 1992.
- [Gibson92] G. A. Gibson, R. H. Patterson and M. Satyanarayanan, "Disk Reads with DRAM Latency", *Third Workshop on Workstation Operating Systems*, Key Biscayne, Florida, April 23-24, 1992.



- [Hac89] A. Hac, "A Distributed Algorithm for Performance Improvement Through File Replication, File Migration, and Process Migration", *IEEE Transactions on Software Engineering* 15, 11 (November 1989), 1459-1470.
- [Harker81] J. M. Harker, D. W. Brede, R. E. Pattison, G. R. Santana and L. G. Taft, "A Quarter Century of Disk File Innovation", *IBM Journal of Research and Development* 25, 5 (September 1981), 677-689.
- [Henderson89] R. L. Henderson and A. Poston, "MSS II and RASH: A Mainframe UNIX Based Mass Storage System with a Rapid Access Storage Hierarchy File Management System", *Winter USENIX 1989*, January 1989, 65-83.
- [Holland92] M. Holland and G. Gibson, "Parity Declustering for Continuous Operation in Redundant Disk Arrays", *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, October 12-15, 1992, 23-35.
- [IBM0661] *IBM 0661 Disk Drive Product Description--Model 371*, IBM, July 11, 1989.
- [Johnson84] O. G. Johnson, "Three-Dimensional Wave Equation Computations on Vector Computers", *Proceedings of the IEEE* 72, 1 (January 1984).
- [Jones89] A. L. Jones, *SSD is Cheaper than DASD*, Storage Technology Corporation, October 1989.
- [Katz90] R. H. Katz, D. W. Gordon and J. A. Tuttle, "Storage System Metrics for Evaluating Disk Array Organizations", UCB/Computer Science Dpt. 90/611, University of California at Berkeley, December 1990.
- [Katz91] R. H. Katz, T. E. Anderson, J. K. Ousterhout and D. A. Patterson, "Robo-line Storage: Low Latency, High Capacity Storage Systems over Geographically Distributed Networks", UCB/Computer Science Dpt. 91/651, University of California at Berkeley, September 1991.
- [Kim86] M. Y. Kim, "Synchronized Disk Interleaving", *IEEE Transactions on Computers* C-35, 11 (November 1986), 978-988.

- [Lee91a] E. K. Lee and R. H. Katz, "An Analytic Performance Model of Disk Arrays and its Applications", UCB/Computer Science Dpt. 91/660, University of California at Berkeley, 1991.
- [Lee91b] E. K. Lee and R. H. Katz, "Performance Consequences of Parity Placement in Disk Arrays", *Proceedings of the 4rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, April 1991, 190-199.
- [Livny87] M. Livny, S. Khoshafian and H. Boral, "Multi-Disk Management Algorithms", *Performance Evaluation Review, Special Issue 15, 1* (May 1987), 69-77. ACM SIGMETRICS 1987.
- [Menon87] J. Menon and M. Hartung, "The IBM 3990 Model 3 Disk Cache", RJ 5994 (S9593), IBM, December 9, 1987.
- [Metrum91] *RSS-600 Rotary Storage System Product Information*, Metrum, 1991.
- [Miller91a] E. L. Miller and R. H. Katz, "Input/Output Behavior of Supercomputing Applications", *Proceedings of Supercomputing '91*, 1991, 567-576.
- [Miller91b] E. L. Miller, "File Migration on the Cray Y-MP at the National Center for Atmospheric Research", UCB/Computer Science Dpt. 91/638, University of California at Berkeley, June 1991.
- [Montgomery91] J. B. Jones(Jr.) and T. Liu, editors. "RAID: A Technology Poised for Explosive Growth", Report DJIA: 2902, Montgomery Securities, December 17, 1991.
- [Muntz90] R. R. Muntz and J. C. S. Lui, "Performance Analysis of Disk Arrays under Failure", *Proceedings of the 16th Conference on Very Large Data Bases*, 1990. VLDB XVI.
- [Myers86] G. J. Myers, A. Y. C. Yu and D. L. House, "Microprocessor Technology Trends", *Proceedings of the IEEE* 74, 12 (December 1986), 1605-1622.
- [Nelson88] M. N. Nelson, B. B. Welch and J. K. Ousterhout, "Caching in the Sprite Network File System", *ACM Transactions on Computer Systems* 6, 1 (February 1988), 134-154.

- [Ousterhout88] J. K. Ousterhout, A. Cherson, F. Dougliis and M. Nelson, "The Sprite Network Operating System", *IEEE Computer* 21, 2 (February 1988), 23-36.
- [Ousterhout89] J. K. Ousterhout and F. Dougliis, "Beating the I/O Bottleneck: A Case for Log-Structured File Systems", *SIGOPS* 23, 1 (January 1989), 11-28.
- [Patterson88] D. A. Patterson, G. Gibson and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *International Conference on Management of Data (SIGMOD)*, June 1988, 109-116.
- [Patterson90] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.
- [Rosenblum91] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System", *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.
- [Salem86] K. Salem and H. Garcia-Molina, "Disk Striping", *Proceedings of the Second International Conference on Data Engineering*, 1986, 336-342.
- [Smith81] A. J. Smith, "Optimization of I/O Systems by Cache Disk and File Migration: A Summary", *Performance Evaluation* 1, 3 (November 1981), 249-262.
- [Smith85] A. J. Smith, "Disk Cache-Miss Ratio Analysis and Design Considerations", *ACM Transactions on Computer Systems* 3, 3 (August 1985), 161-203.
- [TPCB90] *TPC Benchmark B Standard Specification*, Transaction Processing Performance Council, August. 23, 1990.
- [Tan89] E. Tan and B. Vermeulen, "Digital audio tape for data storage", *IEEE Spectrum*, October 1989, 34-38.
- [Thanhardt88] E. Thanhardt and G. Harano, "File Migration in the NCAR Mass Storage System", *Proceedings of the Ninth IEEE Symposium on Mass Storage Systems*, October 1988.

[Vermeulen89] B. Vermeulen, "Helical Scan and DAT--a Revolution in Computer Tape Technology", *Systems Design and Networks Conference (SDNC)*, May 1989, 79-86.

[Weikum90] G. Weikum, P. Zabback and P. Scheuermann, *Dynamic File Allocation in Disk Arrays*, ETH Zurich, December 1990.

---

## Chapter 2

### Previous Benchmarks

---

#### 2.1. I/O Benchmarks

Chapter 1 highlighted some trends and developments in disks, disk arrays, file caches, solid state disks, tapes, and file systems which create new challenges for the evaluation of storage systems. As a result, benchmarks used in the evaluation process must evolve to comprehensively stress these new I/O systems. For example, because disk arrays are able to service many I/Os at the same time, benchmarks need to issue many simultaneous I/Os if they hope to stress a disk array. Caches create distinct performance regions based on the capacity used by a program, so benchmarks likewise should measure these different performance regions.

In this chapter, I list standards that I will use to critique I/O benchmarks. I then review, run, and evaluate I/O benchmarks in use today.

## 2.2. The Ideal I/O Benchmark

In this thesis, I define I/O benchmarks as measuring the data I/O performance seen by a program issuing reads and writes. Specifically, I am *not* using I/O benchmarks to measure the performance of file system commands, such as deleting files, making directories, or opening and closing files. While these are perfectly valid and important metrics, they are more a measure of the operating system and processor speed than they are of the storage components.

It is unfortunate that most people use trivial benchmarks when purchasing and evaluating I/O systems. These include, for example, the time it takes to write 1 MB to disk, the average disk access time, or the raw disk transfer rate. These metrics are similar to the CPU clock rate in processor performance evaluation in that they are better than nothing but do not translate easily into performance that an end user can see. To correct this blind acceptance of trivial benchmarks, I list six desirable characteristics of I/O benchmarks in this section.

First, a benchmark should help system designers and users understand why a system performs as it does. This is because computer architects and operating system programmers need to have benchmarks to evaluate design changes and isolate reasons for poor performance. In addition, users should be also able to use benchmarks to understand the optimal ways of using their machines. For instance, if a user wanted to have his application fit within the file cache, the ideal I/O benchmark should be able to provide information on the size of a machine's file cache. This criterion may require reporting results for several different workloads, which would enable the user to compare these results. These multiple workloads should require little human interaction to run.

Second, to maintain the focus of measuring and understanding I/O systems, the performance of an I/O benchmark should be limited by the I/O devices. The most intuitive test of being I/O-limited is the following: if the speedup resulting from taking out all I/Os from an application is greater than the speedup resulting from taking out all CPU operations (leaving only I/O), then the application is I/O-limited. Unfortunately, this test is quite hard to perform in practice—almost any non-trivial application will not function when its I/O is eliminated. Instead, I test for how I/O-limited an application is by measuring the fraction of time it spends in doing reads and writes.

Third, the ideal I/O benchmark should scale gracefully over a wide range of current and future machines. Without a well planned scaling strategy, I/O benchmarks quickly become obsolete as machines evolve. For instance, *I/OStone* tries to exercise a system's entire memory hierarchy, but touches only 1 MB of user data. Perhaps at the time *I/OStone* was written, 1 MB was a lot of data, but, of course, this is no longer true. Another example of an I/O benchmark's need to scale is provided by disk arrays. As mentioned above, disk arrays allow multiple I/Os to be in progress simultaneously. But, because most current I/O benchmarks do not scale the number of processes issuing I/O, they are unable to properly stress disk arrays. Unfortunately, it is difficult to find widespread agreement on a scaling strategy, especially for benchmarks intended for a broad range of audiences.

Fourth, a good I/O benchmark should allow fair comparisons across machines and should include two aspects. First, a fair comparison across machines should be able to be made for I/O workloads identical to the benchmark. However, since users rarely have the same workload as a standard benchmark, the results from a benchmark should predict performance for workloads that differ from itself.

Fifth, the ideal I/O benchmark should be relevant to a wide range of applications. It is certainly easier to target a benchmark to a specific audience, and benchmarks that do a good job

representing their target applications are invaluable for those applications. But, benchmarks usable by many audiences would be better still.

Finally, in order for results to be meaningful, benchmarks should be tightly specified. Results should be reproducible by general users; optimizations that are allowed and disallowed should be explicitly stated; the machine environment on which the benchmarking takes place should be well-defined and reported (CPU type and speed, operating system, compiler, network, disk, other load on the system); the starting state of the system (file cache state, data layout on disk) should be well-defined and consistent, and so on.

In summary, the six characteristics of the ideal I/O benchmark are as follows: it should help the evaluator understand system performance; its performance should be I/O limited; it should scale gracefully over a wide range of current and future machines; it should allow fair comparisons across machines; it should be relevant to a wide range of applications; and it should be tightly specified.

### 2.3. System Platforms

System Name	SPARCstation 1+	DECstation 5000/200	HP 730
Year Released	1989	1990	1991
CPU	SPARC	MIPS R3000	PA-RISC
SPECmarks	8.3	19.9	76.8
Disk System	CDC Wren IV	3 disk (Wren) RAID 0	HP 1350SX
I/O Bus	SCSI-I	SCSI-I	Fast SCSI-II
Memory Size	28 MB	32 MB	32 MB
Mem. Bus Peak Speed	80 MB/s	100 MB/s	264 MB/s
Mem. Bus Sustained Speed	25-30 MB/s	???	125 MB/s
Operating System	SunOS 4.1	Sprite LFS	HP/UX 8.07

**Table 2.1: System platforms.** This table shows the three systems on which benchmarks were run. The DECstation [DECstation90] uses a three disk RAID disk array [Patterson88] with a 16 KB striping unit [Chen90] and is configured without redundancy. The SPECmark rating is a measure of the processor speed; ratings are relative to the speed of a VAX 11/780. The full name of the HP 730 is the HP Series 700 Model 730 [HP730].



In this chapter, I run standard I/O benchmarks on three systems. Additional systems are tested in Chapters 4 and 5 (see Table 4.1 on page 64). All systems in this chapter are high-performance workstations with differing I/O systems; Table 2.1 summarizes their characteristics. Note that these computers were introduced in different years—this study is not meant to be a competitive market analysis of the competing products.

In order to better understand these benchmarks, I modified their software slightly. For example, I compiled in special I/O routines that traced I/O activity. To accomplish this, I used publicly available code for as many programs as possible. In general, I used GNU (Gnu's Not Unix) code developed by the Free Software Foundation. To make results directly comparable between machines for benchmarks that used the compiler, I took the same step as Ousterhout [Ousterhout90] in having the GNU C compiler generate code for an experimental CPU called SPUR [Hill86].

#### 2.4. Overview of Current I/O Benchmarks

In this section, I describe, critique, and run five common benchmarks used in I/O system evaluation: Andrew, TPC-B, Sdet, Bonnie, and IOStone. Table 2.2 contains information about how to obtain these benchmarks. I categorize them into two classes: application benchmarks and synthetic benchmarks.

Benchmark	Contact	E-mail address
SPEC SDM	National Computer Graphics Corp.	spec-ncga@cup.portal.com
TPC	Shanley Public Relations	shanley@cup.portal.com
Bonnie	Tim Bray	tbray@watsol.Waterloo.EDU
IOStone	Jeffrey Becker	becker@iris.ucdavis.EDU
LADDIS	Bruce Keith	spec-preladdis-beta-test@riscee.pko.dec.com

Table 2.2: List of contacts for various benchmarks.

## 2.4.1. Application Benchmarks

Application benchmarks use standard programs, such as compilers, utilities, editors, and databases, in various combinations to produce a workload. Each benchmark targets a single application area, such as transaction processing or system development. Although these benchmarks usually do a good job of accurately representing their target application area, as will be seen, they are often not I/O limited.

### 2.4.1.1. Andrew

Andrew was designed at Carnegie Mellon University to be a file system benchmark for comparing the Andrew File System against other file systems [Howard88]. It was originally meant to act only as a convenient yardstick for measuring file systems and not necessarily as a representative workload for benchmarking. Despite this intent, however, it has become a widely used, de facto benchmarking standard [Ousterhout90].

Andrew is meant to represent the workload generated by a typical set of software system developers. It copies a file directory hierarchy, examines and reads the new copy, then compiles the copy. The file directory contains 70 files totaling 0.2 MB. Carnegie Mellon's experience in 1987 suggests the load generated roughly equals that generated by five users.

System	Copy Phase		Compile Phase		Total	
	Time	% I/O	Time	% I/O	Time	% I/O
SPARCstation 1+	82 sec.	4%	137 sec.	7%	219 sec.	6%
DECstation 5000	20 sec.	10%	67 sec.	3%	87 sec.	4%
HP 730	17 sec.	27%	28 sec.	6%	45 sec.	13%

**Table 2.3: Results from the Andrew benchmark.** This table shows the results obtained from running the Andrew Benchmark. Andrew is divided into two sections: the *copy phase*, consisting of the copy, examination, and reading stages; and the *compile phase*. In each column, the percentage of time spent performing reads and writes is listed. Each of these numbers represents the average of three runs; each run starts with an empty file cache. Note the small percentage of time spent in I/O.

In Table 2.3, I list results derived from Andrew on the three system platforms. As in [Ousterhout90], I divide Andrew into two sections: the *copy phase*, consisting of the copy, examination, and reading stages; and the *compile phase*. Note that on all machines Andrew spends only 6%-13% actually doing data reads and writes. Also note that the HP 730, which has the fastest CPU, spends a higher fraction of time in I/O than the others, a result that supports my contention that systems with faster and faster CPUs will become more and more I/O limited.

#### 2.4.1.2. TPC-B

TPC-B measures transaction processing performance for a simple, database update [TPCB90]. The first version of this benchmark, TP1, first appeared in 1985 [Anon85] and quickly became the de facto standard in benchmarking for transaction processing systems.

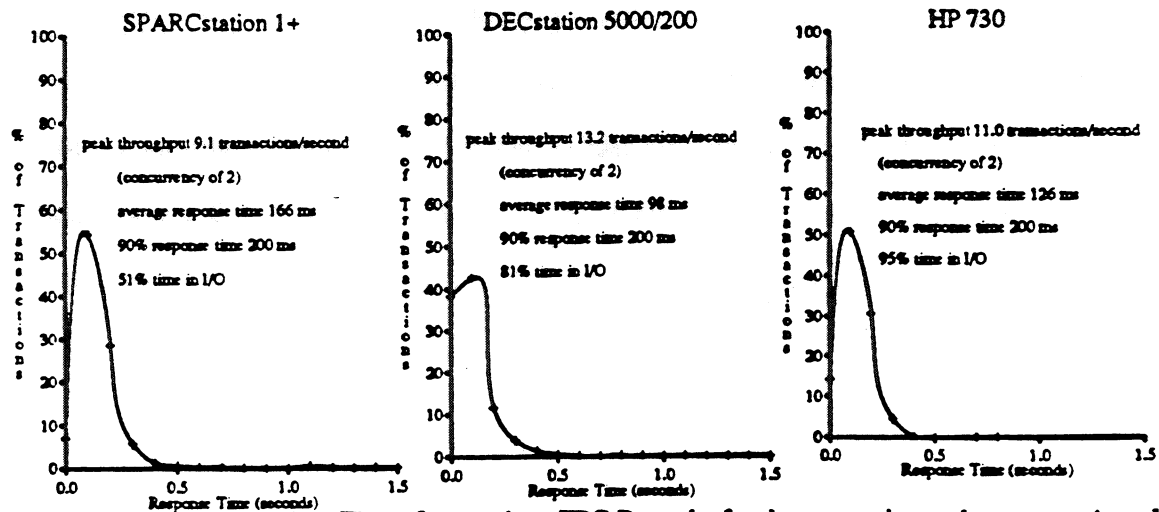


Figure 2.1: TPC-B results. These figures show TPC-B results for three experimental systems. As a database program, Seltzer's simple transaction processing library LIBTP [Seltzer92] were used. Due to a deadlocking bug in the software package, however, concurrencies higher than two were not run. This is reflected by response times much faster than those required by TPC-B.

TPC-A<sup>1</sup> [TPCA89] and TPC-B [TPCB90] are more tightly specified versions of TP1 and have replaced it as the standard transaction processing benchmark. As a transaction processing benchmark, TPC-B not only measures the machine supporting the database but also the database software.

TPC-B repeatedly performs *Debit-Credit* transactions, each of which simulates a typical change to an account on a bank's database, which consists of customer accounts, bank branches, and tellers. Using a random customer request, a Debit-Credit transaction reads and updates the necessary account, branch, and teller balances. Requests are generated by a number of simulated customers, each requesting transactions as quickly as possible.

TPC-B's main metric is maximum throughput measured in transactions-per-second, qualified by a response time threshold demanding that 90% of all transactions be completed within two seconds. TPC-B also reports the price of the system and the required storage. The number of accounts, branches, and tellers specified by TPC-B is proportional to throughput—for each additional transaction-per-second of performance reported, the test system's database must add 10 MBs of account information. Using a database of this size, TPC-B reports a graph of throughput versus the average number of outstanding requests as well as a histogram of response times for the maximum throughput. In Figure 2.1, I show TPC-B's response time characteristics on the three systems, using Seltzer's simple, transaction supporting package LIBTP [Seltzer92].

---

<sup>1</sup>The main difference between TPC-A and TPC-B is the presence of real terminals. TPC-A demands that the test be done with actual terminals providing input at an average rate of one request every 10 seconds. TPC-B generates requests with internal drivers running as fast as possible. This thesis discusses only TPC-B.

### 2.4.1.3. Sdet

The System Performance Evaluation Cooperative (SPEC) was founded in 1988 to establish independent standard benchmarks [Scott90]. Their first set of benchmarks, SPEC Release 1, primarily measures CPU performance. Their second set of benchmarks, System Development Multi-tasking (SDM) Suite, measures overall system performance for software development and research environments. SDM consists of two benchmarks, Sdet [Gaede81, Gaede82] and Kenbus1 [McDonnell87]. Sdet and Kenbus1 are quite similar in benchmarking methodology; their main difference is the specific mix of user commands. Because Sdet does much more I/O than Kenbus1, I discuss only Sdet.

Sdet's workload consists of a number of *scripts* running concurrently. Each script contains a list of user commands in random order. These commands are taken from a typical software-development environment and include editing, text formatting, compiling, file creating

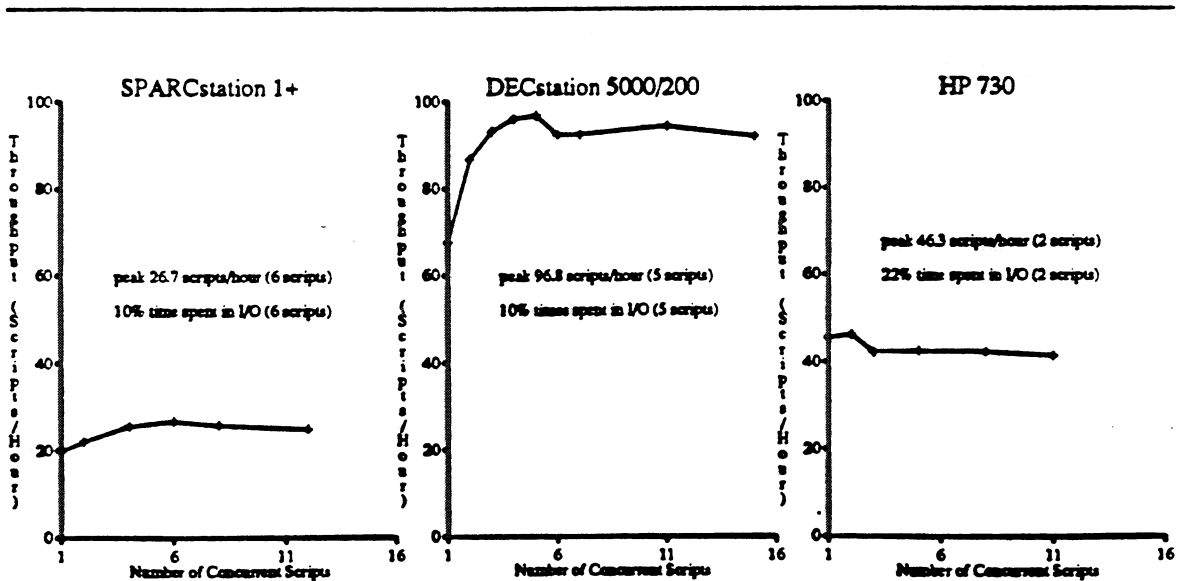


Figure 2.2: Sdet results. These figures show results from the SPEC SDM benchmark Sdet. Sdet varies the number of scripts running simultaneously but spends little time in I/O.

and deleting, as well as miscellaneous other UNIX utilities [SPEC91]. Sdet increases the number of scripts running concurrently until it reaches the system's maximum throughput, which is measured as the script-completion rate (scripts per hour). Sdet reports this maximum rate, along with a graph of throughput versus script concurrency (Figure 2.2). Once again, only a small percentage of time, 10%-22%, is spent in I/O.

## 2.4.2. Synthetic Benchmarks

Unlike application benchmarks that use standard programs, which in turn issue I/O, synthetic benchmarks exercise an I/O system by directly issuing read and write commands. By issuing reads and writes directly, synthetic benchmarks are able to generate more I/O-intensive workloads. However, synthetic benchmarks often yield less convincing results because they, unlike application benchmarks, do not perform useful work. I review four synthetic benchmarks here. Two of the most popular benchmarks are Bonnie and IOStone. I also create a third one to demonstrate a typical scientific I/O workload. Finally, I review an emerging benchmark for network file systems called LADDIS.

### 2.4.2.1. Bonnie

Bonnie measures I/O performance on a single file for a variety of simple workloads. One workload reads sequentially the entire file one character at a time while another writes the file

Workload	SPARCstation 1+		DECstation 5000		HP 730	
	KB/s	% I/O	KB/s	% I/O	KB/s	% I/O
Sequential Char Write	229	1%	300	23%	1285	2%
Sequential Block Write	558	63%	663	86%	1777	84%
Sequential Block Rewrite	230	74%	297	86%	604	87%
Sequential Char Read	193	2%	253	40%	995	15%
Sequential Block Read	625	63%	781	82%	2023	87%
Random Block Read	31 IOs/sec	78%	32 IOs/sec	86%	39 IOs/sec	95%

Table 2.4: Results from Bonnie. This table shows results from running Bonnie. A 100 MB file was used for all runs.

one character at a time. Other workloads exercise block-sized sequential reads, writes, or reads followed by writes (rewrites). The final workload uses three processes to simultaneously issue random I/Os. The size of the file is set by the evaluator and should be several times larger than the system's file cache, thus preventing the entire file from fitting in the cache. For each workload, Bonnie reports throughput, measured in KB per second or I/Os per second, and CPU utilization. I show results for the three systems in Table 2.4. Most of Bonnie's workloads are I/O-limited, though the character reads and writes are CPU-limited.

#### 2.4.2.2. IOStone

IOStone is a synthetic I/O benchmark [Park90] based on system traces of Unix minicomputers and workstations [Ousterhout85, Hu86] and IBM mainframes [Smith78, Smith81]. Using 400 files totaling 1 MB, IOStone reads and writes data in patterns which approximate the locality found in [Ousterhout85]. One process performs all the accesses—no I/O parallelism is present. IOStone reports a single throughput result, measured in IOStones per second (Table 2.5). Though the file caches of the HP 730 and the SPARCstation 1+ are large enough to contain all of the data of IOStone, HP/UX and SunOS limit the number of files present in the file cache to approximately 300. Since IOStone touches 400 files, many references access the disk. IOStone runs much faster on the DECstation 5000, because Sprite imposes no such limit, and all accesses can be satisfied from the file cache.

System	IOStones/Second	Percent Time Spent in I/O
SPARCstation 1+	11,002	61%
DECstation 5000	50,905	26%
HP 730	20,409	81%

**Table 2.5: Results from IOStone.** This table shows results from running IOStone. HP/UX and SunOS limit the number of files resident in the file cache, so many references need to access the disk. Because Sprite allows all IOStone data files to reside in the file cache, its performance is 2-4 times better, and much less I/O limited, than the other systems.

### 2.4.2.3. Sample Scientific Workload

Andrew, SDM, IOStone, and Bonnie all target system development or workstation environments. Other application areas, such as scientific or supercomputing code, have substantially different workload characteristics [Miller91]. Typical scientific applications generally touch much more data and use much larger request sizes than workstation applications. To illustrate I/O performance for a supercomputing environment, I define two simple workloads: a large file read workload, which reads a 100 MB file in 128 KB units, and a large file write workload, which writes a 100 MB file in 128 KB units. In Table 2.6, I show results for the three system platforms.

### 2.4.2.4. LADDIS

Network file systems provide file service to a set of *client* computers, connected by a network. The computer providing this file service is called the *server*. One popular protocol for network file service is Sun Microsystem's NFS [Sandberg85]. In 1989, Shein, Callahan, and Woodbury created NFSStone, a synthetic benchmark to measure NFS performance [Shein89]. NFSStone generated a series of NFS file requests from a single client to stress and measure server performance. These operations included reads, writes, and various other file operations such as examining a file. The exact mix of operations was patterned after a study done by Sun [Sandberg85]; the sizes of the files were patterned after the study done in [Ousterhout85].

System	Large File Reads		Large File Writes	
	Throughput	% I/O	Throughput	% I/O
SPARCstation 1+	0.64 MB/s	99%	0.64 MB/s	98%
DECstation 5000	0.82 MB/s	92%	1.28 MB/s	98%
HP 730	1.98 MB/s	100%	1.86 MB/s	98%

**Table 2.6: Results from two sample scientific workloads.** This table shows the results obtained from running two simple workloads typical of scientific applications. *Large File Reads* consists of sequentially reading a 100 MB file in 128 KB units. *Large File Writes* consists of sequentially writing a 100 MB file in 128 KB units.



Later, Legato Systems refined NFSStone and called it NHFSSStone to avoid copyright infringement. Nevertheless, NFSStone and NHFSSStone had several problems: one client could not always fully stress a file server; different versions of the benchmarks abounded; file and block sizes were not realistic; and only SunOS clients could run them.

In 1990, seven companies, joined forces to create a NFS benchmark capable of stressing even the most powerful file servers. The result was *LADDIS*, named after the seven companies (Legato, Auspex, Digital Equipment Corporation, Data General, Interphase, and Sun). It will be incorporated as part of the SPEC SFS (System Level File Server) Suite of benchmarks. *LADDIS* is based on NHFSSStone but, unlike NHFSSStone, runs on multiple, possibly heterogeneous, clients and networks [Nelson92, Levitt92]. Like NHFSSStone, *LADDIS* is a synthetic benchmark with a set mix of operations. As of this writing, *LADDIS* is still under development. Its current form includes default percentages for the mix of operations, although it allows the evaluator to modify that mix. In its final release, SPEC will likely standardize the operation mix. Currently, *LADDIS* can vary many parameters. For example, besides the percentage of each operation in the workload, *LADDIS* gives the evaluator the freedom to change the number of clients issuing requests to the server, the rate at which each client issues requests, the total size of all files, and the block size of I/O requests. Reads are weighted to produce 85% block-sized requests and 15% partial-block requests; Writes are weighted to produce 50% of each. Moreover, *LADDIS* touches many files, and each operation's data files are chosen randomly.

The reporting philosophy of *LADDIS* is quite similar to that of TPC-B. The preferred metric is a throughput (NFS operations per second) versus response time graph. As a more compact form, users may report the maximum throughput possible subject to a response time constraint of 50 ms. Like TPC-B, *LADDIS* scales according to the reported throughput—for every 100 NFS operations/second of reported throughput, capacity must increase by at least 1 GB—although it does not yet specify how to scale the rest of the workload parameters.

LADDIS is expected to be released by SPEC in 1993.

## 2.5. Critique of Current Benchmarks

In applying my list of benchmark goals from Section 2.2 to current I/O benchmarks, it can be seen that there is much room for improvement. I show a qualitative evaluation of today's I/O benchmarks in Figure 2.3 and make the following observations:

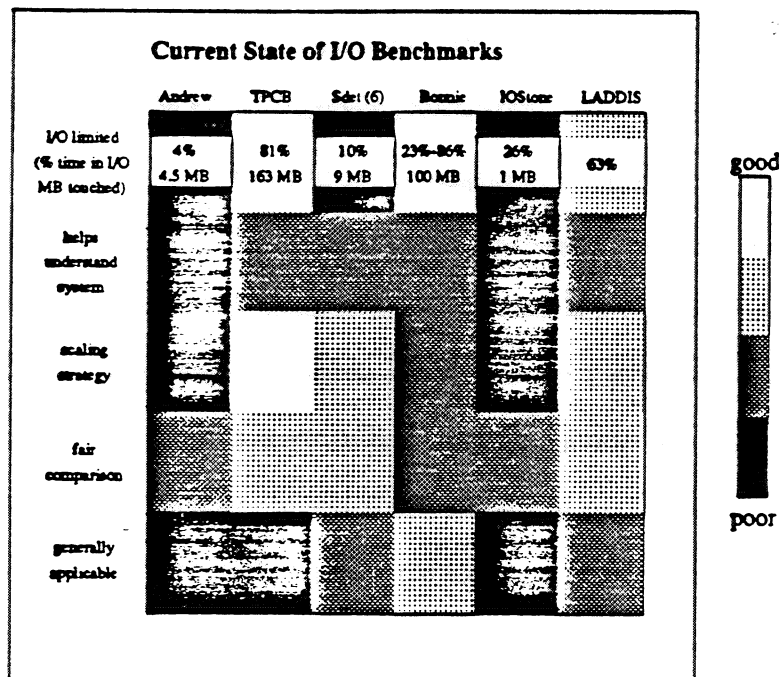


Figure 2.3: Current state of I/O benchmarks. This figure shows a qualitative evaluation of benchmarks used today to evaluate I/O systems. It can be seen that several are not I/O bound and that most do not provide understanding of the system, lack a well-defined scaling strategy, and are not generally applicable. The percent time spent in I/O was measured on the DECstation 5000/200 of Table 2.1. LADDIS was not available for execution as of this writing, but a pre-release beta version spends 63% of its execution time doing reads and write; the rest of the time is spent in other NFS operations, such as lookup (17%) and getattr (6%).

- *Many I/O benchmarks are not I/O limited.* On the DECstation 5000/200, Andrew, Sdet<sup>2</sup>, and IOStone spend a quarter or less of their time doing I/O. Furthermore, many of the benchmarks touch very little data; IOStone touches only 1 MB of user data; Andrew touches only 4.5 MB. The best of the group is Bonnie, but even it has some tests that were CPU-bound.
- *Today's I/O benchmarks do not help in understanding system performance.* Andrew and IOStone give only a single, bottom-line performance result. TPC-B and Sdet fare somewhat better because they help the user understand system response under various loads, while Bonnie begins to help the user understand performance by running six workloads. These workloads show the differences in performance between reads versus writes and block versus character I/O, but do not vary other aspects of the workload, such as the number of I/O's occurring in parallel.
- *Many of today's I/O benchmarks have no scaling strategy.* For example, several made no provision for adjusting the workload to stress machines with larger file caches, for example. Without a well-defined scaling strategy, I/O benchmarks quickly grow obsolete. Several exceptions are noteworthy. TPC-B has an extremely well-defined scaling strategy, made possible by TPC-B's narrow focus on debit-credit style transaction processing and the widespread agreement on how databases change with increasing throughput. Sdet also has a superior scaling strategy, which varies the number of simultaneously active scripts until the peak performance is achieved. Automatically scaling certain aspects of a workload represents a major improvement over single workload benchmarks. However, Sdet does not scale any other aspects of the benchmark, such as request size or read/write ratio. LADDIS, when formally defined, will likely have a scaling strategy similar to Sdet: it will probably

---

<sup>2</sup> This refers to Sdet running at a peak throughput concurrency level of 6.

scale a few workload parameters, such as disk space or number of clients, but will leave other parameters inflexible.

- *Today's I/O benchmarks make fair comparisons for workloads identical to the benchmark, but do not help in drawing conclusions about the relative performance of machines for other workloads. Ideally, the results from a benchmark could be applied to a wider range of workloads.*
- *Today's I/O benchmarks focus on a narrow range of application range. For example, TPC-B is intended solely for benchmarking debit-credit, transaction processing systems.*

The generally poor state of I/O benchmarks suggests an urgent need for new benchmarks.

## 2.6. References

- [Anon85] Anon and *et al.*, "A Measure of Transaction Processing Power", *Datamation*, 31, 7 (April 1, 1985), 112-118.
- [Bechtolsheim90] A. V. Bechtolsheim and E. H. Frank, "Sun's SPARCstation 1: A Workstation for the 1990s", *Proceedures of the IEEE Computer Society International Conference (COMPCON)*, Spring 1990, 184-188.
- [Chen90] P. M. Chen and D. A. Patterson, "Maximizing Performance in a Striped Disk Array", *Proceedings of the 1990 ACM SIGARCH Conference on Computer Architecture*, Seattle WA, May 1990, 322-331.
- [DECstation90] *DECstation 5000 Model 200 Technical Overview*, Digital Equipment Corporation, 1990.
- [Gaede81] S. Gaede, "Tools for Research in Computer Workload Characterization", *Experimental Computer Performance and Evaluation*, 1981. D. Ferrari, M. Spadoni, eds..
- [Gaede82] S. Gaede, "A Scaling Technique for Comparing Interactive System Capacities", *13th International Conference on Management and Performance Evaluation of Computer Systems*, 1982, 62-67. CMG 1982.

- [HP730] *HP Apollo Series 700 Model 730 PA-RISC Workstation*, Hewlett-Packard, 1992.
- [Hill86] M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, R. H. Katz, J. K. Ousterhout and D. A. Patterson, "Design Decisions in SPUR", *IEEE Computer* 19, 11 (November 1986).
- [Horning91] R. Horning, L. Johnson, L. Thayer, D. Li, V. Meier, C. Dowdell and D. Roberts, "System Design for a Low Cost PA-RISC Desktop Workstation", *Procedures of the IEEE Computer Society International Conference (COMPCON)*, Spring 1991, 208-213.
- [Howard88] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham and M. J. West, "Scale and Performance in a Distributed File System", *ACM Transactions on Computer Systems* 6, 1 (February 1988), 51-81.
- [Hu86] I. Hu, "Measuring File Access Patterns in UNIX", *Performance Evaluation Review* 14, 2 (1986), 15-20. ACM SIGMETRICS 1986.
- [Levitt92] J. Levitt, "Better Benchmarks are Brewing", *Unix Today!*, January 20, 1992.
- [McDonell87] K. J. McDonell, "Taking performance evaluation out of the stone age", *Proceedings of the Summer Usenix Technical Conference*, Phoenix, Arizona, June 1987, 407-417.
- [Miller91] E. L. Miller and R. H. Katz, "Input/Output Behavior of Supercomputing Applications", *Proceedings of Supercomputing '91*, 1991, 567-576.
- [Nelson92] B. Nelson, B. Lyon, M. Wittle and B. Keith, "LADDIS--A Multi-Vendor & Vendor-Neutral NFS Benchmark", *UniForum Conference*, January 1992.
- [Nielsen91] M. J. K. Nielsen, "DECstation 5000 Model 200", *Procedures of the IEEE Computer Society International Conference (COMPCON)*, Spring 1991, 220-225.

- [Ousterhout85] J. K. Ousterhout, H. Da Costa and *et al.*, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System", *Operating Systems Review* 19, 5 (December 1985), 15-24. Proceedings of the 10th Symp. on Operating System Principles.
- [Ousterhout90] J. K. Ousterhout, "Why aren't operating systems getting faster as fast as hardware?", *Proceedings USENIX Summer Conference*, June 11-15, 1990, 247-256.
- [Park90] A. Park and J. C. Becker, "IOStone: A synthetic file system benchmark", *Computer Architecture News* 18, 2 (June 1990), 45-52.
- [Patterson88] D. A. Patterson, G. Gibson and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *International Conference on Management of Data (SIGMOD)*, June 1988, 109-116.
- [SPEC91] *SPEC SDM Release 1.0 Manual*, System Performance Evaluation Cooperative, 1991.
- [Sandberg85] R. Sandberg, D. Goldbert, S. Kleiman, D. Walsh and B. Lyon, "Design and Implementation of the Sun Network Filesystem", *Summer 1985 Usenix Conference*, 1985.
- [Scott90] V. Scott, "Is Standardization of Benchmarks Feasible?", *Proceedings of the BUSCON Conference*, Long Beach, CA, Feb 14, 1990, 139-147.
- [Seltzer92] M. Seltzer and M. Olson, "LIBTP: Portable, Modular Transactions for UNIX", *USENIX 1992?*, January 1992.
- [Shein89] B. Shein, M. Callahan and P. Woodbuy, "NFSStone--A Network File Server Performance Benchmark", *Proceedings of the USENIX Summer Technical Conference 1989*, , 269-275.
- [Smith78] A. J. Smith, "Sequentiality and Prefetching in Database Systems", *ACM Transactions on Database Systems* 3, 3 (1978), 223-247.
- [Smith81] A. J. Smith, "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms", *IEEE Transactions on Software Engineering SE-7*, No. 4 (1981), 403-417.

[TPCA89] *TPC Benchmark A Standard Specification*, Transaction Processing Performance Council, November 10, 1989.

[TPCB90] *TPC Benchmark B Standard Specification*, Transaction Processing Performance Council, August. 23, 1990.

---

## Chapter 3

# Workload Model

---

### 3.1. Overview

The core of any benchmark used in performance evaluation is the workload model. The goals of a good workload model are flexibility, generality, and simplicity [Ferrari72, Ferrari84]. Flexibility requires the model to allow for the selective tuning of individual workload features. Generality requires that a wide range of applications and programs be modeled accurately. Accuracy here refers to how closely the workload model is able to duplicate the performance of an application. And finally, simplicity requires that an application be defined easily in terms of the workload model. This chapter reviews I/O workload generators used in recent



research, describes several I/O tracing studies, discusses in detail *Willy*<sup>1</sup>, the workload generator used in this thesis, and relates how current I/O benchmarks can be accurately modeled by *Willy*.

### 3.2. I/O Workloads Used in Research

I/O workload generators outside the realm of benchmarking function primarily as aids in doing research on I/O architectures. These workloads fall into two classes: *open-system* workloads and *closed-system* workloads [Denning78].

Open-system workloads issue I/Os independently of I/O completions according to a fixed interarrival probability distribution. After one I/O is issued, the arrival generator waits a period of time, determined by the interarrival probability distribution, then repeats the process. Because new I/Os are issued independently of I/O completions, arbitrarily many I/Os could be in progress. When I/Os arrive faster than they finish, the number of I/Os in progress increases without limit, and the system is said to be in *saturation*.

The most common interarrival distribution is an exponential waiting time. An exponential waiting time between successive I/Os leads to I/Os arriving as what is called a Poisson process. Wilhelm uses this to derive a performance model for a disk system [Wilhelm77], while Geist and many others use this workload to analyze various disk-scheduling policies [Geist87, Oney75, Gotlieb73, Teorey72, Hofri80, Coffman72]. Arrival rate is usually the only workload parameter varied in these studies, though other parameters such as transfer size could also be varied.

In contrast to open-system workloads, closed-system ones issue I/Os by using a fixed number of I/O generators. Each generator issues an I/O, *waits until the I/O is completed*, pauses for another period of time, then repeats. Unlike open systems, new I/Os generated by a closed systems depend on the completion of the last request from that generator. Stone argues that this

---

<sup>1</sup>The name *Willy* is derived from the word *workload*.

method more closely approximates the way real computer systems operate [Stone72], since real systems usually have a fixed number of users who cannot issue their next I/O until their last I/O finishes. If the waiting time between an I/O completion and the next I/O is fixed at zero, then there will always be the same number of I/Os in the system; this is referred to as the *concurrency* or *load* on the system. A good example of a closed-system workload is Bodnarchuk and Bunt's NFS workload generator [Bodnarchuk91]. Their workload has many parameters, including the distribution of reads and writes and other NFS operations, read and write sizes, and which files to reference. The Solbourne Filesystem Benchmarks also use a closed-system workload model [Taylor90], the parameters for which include concurrency and request size. Chen and Lee also use a closed system workload in their studies on the disk array striping unit [Chen90, Lee91]; they add request size as another workload parameter.

### 3.3. I/O Tracing Studies

Another flavor of I/O workload is a trace. Several researchers have gathered I/O traces of various systems, and in this section, I review four important I/O trace studies covering three different computing environments: mainframe database systems, Unix engineering systems, and scientific supercomputing systems.

In 1976 to 1978, Smith and others analyzed traces gathered at a site running an Information Management System/360 (IMS) hierarchical database. The trace was gathered over a seven day period on a 200 MB database [Smith78]. Smith examined sequentiality in the traces to compute the benefits of prefetching. He found that less than 30% of all requests were sequential, much lower than studies done in engineering and scientific environments, though high enough to significantly improve performance through prefetching. Rodriguez-Rosell found that the IMS database generated accesses with little temporal locality [Rodriguez-Rosell76].

In 1985, Ousterhout *et al.* traced the I/Os done on three VAX 11/780 minicomputers at the University of California at Berkeley [Ousterhout85]. These computers were mainly used for

software development, administration, and computer-aided design. The researchers found that over the 2-3 day tracing sample,

- I/O requirements were quite low; each active user required only 300-600 bytes per second of I/O, though bursts of I/O reached 100 KB per second.
- Most files were accessed sequentially in their entirety.
- Most files were short lived; half of all new files were deleted in 5 minutes.
- File caches would be effective in reducing disk accesses.

In 1991, Baker *et al.* performed a follow-up study to Ousterhout's tracing study [Baker91]. These researchers traced the experimental Sprite operating system cluster, which consisted of forty, 10-MIPS workstations at the University of California at Berkeley. These workstations were used in much the same way as the minicomputers in the original Berkeley study: software development, administration, and computer-aided design. The Sprite system, unlike the VAX 11/780, uses a shared, distributed file system spread over a network. In spite of 100 times more CPU power and a distributed file system, many notable trace characteristics were similar to the ones found in the original Berkeley study. For example, files were mostly accessed sequentially in their entirety, and many files were still short-lived. However, Baker *et al.* found that I/O requirements had increased an order of magnitude to 8 KB per second per active user and that large files had grown much larger to be multiple megabytes in size.

In 1991, Miller analyzed seven scientific applications running on a Cray Y-MP supercomputer at NASA Ames [Miller91]. These applications solved mostly computational fluid dynamic problems, such as climate modeling and aerodynamic turbulence. Miller reports a much more I/O intensive environment than did the Berkeley studies. Among the things he found were: data sizes were much larger, ranging between 10 to 200 MB; I/O sizes were larger, ranging from 16 to 400 KB; each application performed large amounts of I/O, from 60 MB to 20 GB; and I/O demands reached 70 MB per second. Like the Berkeley studies, however, the

NASA Ames applications tended to generate bursts of I/O and reference files sequentially.

### 3.4. Willy

In Chapter 2 two types of benchmarks were discussed: synthetic and application. We saw that application benchmarks often have difficulty stressing I/O systems. Application benchmarks are also harder to control than synthetic ones—if a change to a specific part of a workload needs to be made without perturbing the rest of the workload, synthetic benchmarks are much better suited than application workloads. For these reasons, I choose to use a parameterized synthetic benchmark to generate I/Os. This section describes in detail Willy, the workload generator used in this thesis, and how other benchmarks can be described in terms of Willy's parameters.

#### 3.4.1. Workload Parameters

In this section, I describe three main categories of workload parameters: locality, request characteristics, and load.

##### 3.4.1.1. Locality

Locality refers to the data location of each I/O. Due to caches, I/O performance is highly dependent on spatial and temporal locality [Denning70, Smith85]. As a result, the workload model must be able to generate workloads with various spatial and temporal localities. In describing locality, *working set* is used to refer to the data most commonly used by a program. The size of the working set is critical in determining how well caches work for an application. My workload model uses two parameters to characterize locality: *uniqueBytes* and *seqFrac*.

*UniqueBytes* refers to the total number of unique data bytes read or written in a workload—this is the total size of the data. Large values of *uniqueBytes* correspond to workloads with large working sets—any cache in the system will have to be larger to effectively

cache the working set.

Rather than designate a fixed part of the data set as commonly used, Willy models a floating, commonly used data set. While creating a workload, Willy maintains a stack of the most recently used data. To decide which data to access, Willy chooses a depth in the stack and accesses the data at that depth. This method is based on the way most caches operate; most caches choose which data to keep and which to remove based on which was most recently used [Hill87]. This model is known as the LRU stack model [Rau79]. For this reason, workloads generated by Willy should closely mimic the cache behavior of real applications with the same values of `uniqueBytes`. For simplicity, the workload model fixes the average access depth in the LRU stack at half of `uniqueBytes`. Applications that significantly differ from this can compensate by increasing or decreasing `uniqueBytes` to match the average access depth. The distribution of the depth of access in the stack is modeled as a binomial distribution with a mean equal to half of `uniqueBytes`.

`UniqueBytes` only affects temporal locality; another parameter is needed to control spatial locality. The major type of spatial locality in I/O workloads is the presence of sequential accesses. Several of the tracing studies reviewed in Section 3.3 showed large fractions of accesses to be sequential [Ousterhout85, Baker91, Miller91]. *SeqFrac* controls the fraction of sequential accesses (*seqFrac* of 1.0 means all requests are sequential). Addresses for non-sequential accesses are chosen according to the discussion on `hitDepth` about the most recently used data stack. Addresses are chosen randomly for data being accessed for the first time (and hence do not appear in the data stack).

#### 3.4.1.2. Request Characteristics

Two parameters define the characteristics of individual accesses. First, accesses can be either a read or a write. *ReadFrac* determines the fraction of requests that are reads. Each request is chosen to be a read or a write independently; runs of multiple reads or writes are not

deliberately created.

The size of each request is determined by the parameter *sizeMean*. Sizes are distributed according to a binomial distribution with a mean of *sizeMean*. For the sake of simplicity, other types of distributions are not modeled.

### 3.4.1.3. Load

To control the load on the system, I choose the closed-system workload model mentioned in Section 3.2. A closed-system workload model involves two parameters: *processNum* and *cpuThink*. *ProcessNum* is the number of processes running simultaneously, also called concurrency [Chen90]. *CpuThink* is the time each process waits from the time an I/O finishes until it issues the next I/O. Because larger values of *cpuThink* only lessen the fraction of I/O in the workload, I fix *cpuThink* at zero; processes always issue new I/Os immediately after their last I/O completes.

### 3.4.2. Other Workload Issues

Besides these workload parameters, a workload model includes many other issues. In this section, I discuss these issues and the approach Willy takes for each.

The first issue is the performance metric that will be reported by Willy, and in this thesis, the main performance metric will be throughput. Average response time is easily calculated from average throughput in a closed system by Little's Law [Denning78]:

$$\text{throughput} = \frac{\text{sizeMean}}{\text{cpuThink} + \text{responseTime}} \times \text{processNum}$$

The next issue is the number of files accessed in the workload. Due to per-file locking, some file systems, such as Sprite [Rosenblum91] do not allow multiple processes to access a file simultaneously. To allow full use of the file system, my workload generator uses multiple files; the number of files is equal to the number of processes.

In order to keep the read or write property of an access independent of the address of the access, all writes are overwrites; they are not appended at the end of the file. This simplifies the workload model because the entire file is created before the workload starts, thus no new data needs to be allocated.

The next workload issue is the starting layout on disk. Ideally, researchers should start with a freshly made file system on an empty disk, perform a large, standard set of file creations, reads, writes, and file deletions, then run the benchmark. I leave this step for future research.

Address alignment often affects I/O performance. Most I/Os generated by real-world applications are aligned to the block size of the system. In keeping with that, addresses and sizes generated by this workload model are aligned to the block size, usually 4 KB or 8 KB.

The last workload issue addressed here is when to flush the file system cache, if at all. In this thesis, I first prime the file cache before measuring performance. To prime the file cache, I run the workload multiple times and discard the first run. This approach is commonly known as *warm start*.

### 3.5. Modeling Real Applications

In this section, I characterize the I/O patterns of real applications and benchmarks in terms of Willy's parameters. I discuss the benchmarks of Chapter 2 plus an I/O intensive sorting program and some I/O traces of scientific applications from [Miller91]. Table 3.1 shows the parameter values of these applications. Note the wide variation in the parameter values of these applications. Within the variation, however, there are several trends:

- The two file system benchmarks, Andrew and Sdet, touch small amounts data using small requests. Within these benchmarks, reads and writes occur in equal portions and sequentiality is moderate. High degrees of concurrency can be present, depending on how many simultaneous scripts are running.

Application	uniqueBytes (MB)	seqFrac	readFrac	sizeMean (KB)	processNum
Andrew	4.7	.77	.54	3	1
Sdet	8	.48	.56	2.4	4
TPC-B	47.9	.01	.51	4.4	1
ccm	11.6	.91	.51	32	1
gcm	229	1.00	.07	32	1
les	224	.07	.48	325	1
venus	55.2	.43	.65	456	1
sort	234	.99	.56	4.0	1
Minimum	1.8	.01	.07	2.4	1
Maximum	234	1.00	.65	456	4

**Table 3.1: Workload characterization of benchmarks/applications.** This table characterizes several benchmarks and applications in terms of this thesis' workload model. Ccm, gcm, les, and venus are three scientific applications run at NASA Ames Research Center [Miller91]. Ccm, gcm, and venus model atmospheric climates; les models aeronautical turbulence effects. The sort application sorted 4 files totaling 48 MB. Minimum and Maximum show the range of parameter values for these applications. Sdet was run with a concurrency of 5, which yields the maximum throughput on the DECstation 5000 of Table 2.1. More concurrently running scripts would increase the value of processNum, which is the maximum number of simultaneously occurring I/Os. TPC-B was run on the DECstation 5000/200, which delivered 13.2 transactions per second.

- The transaction processing benchmark TPC-B touches a large amount of data using small requests. Requests are mostly non-sequential.
- The scientific applications touch large amounts of data using large requests.

### 3.6. Representativeness of Willy

The most important question in developing a synthetic workload model is the question of representativeness [Ferrari84]. A synthetic workload should have enough parameters such that the performance of the synthetic workload is close to the performance of an application with the same set of parameter values. Of course, given the uncertain path of future computer development, it is impossible to determine *a priori* all the possible parameters necessary to ensure representativeness. Even for current systems, it is possible to imagine I/O workloads that interact with a system in such a way that no synthetic workload (short of a full trace) could duplicate that I/O workload's performance. To show that my workload model captures the



Application	Throughput	Read Response Time	Write Response Time	Average Response Time
Sort	.20 MB/s	19.7 ms	1.6 ms	11.7 ms
Willy	.20 MB/s	20.0 ms	1.9 ms	11.0 ms
TPC-B	.13 MB/s	25.6 ms	1.3 ms	14.0 ms
Willy	.13 MB/s	22.1 ms	1.6 ms	12.3 ms

**Table 3.2: Representativeness of Willy.** This table shows how accurately Willy mimics the performance of two I/O-bound applications, Sort and TPC-B. All runs were done on a DECstation 5000 running Sprite. For each application, the synthetic workload used the parameter values given in Table 3.1.

important features of an I/O workload, this section compares the performance of I/O-bound applications to the performance of Willy with those applications' parameter values.

Out of the four application benchmarks run in this section (Andrew, TPC-B, Sdet, and Sort), only TPC-B and Sort are interesting to compare. Andrew and Sdet spend only a few percent of their running time performing I/O, making them uninteresting to compare. Another way to view this is the following: if an application spends 5% of its running time in I/O, then a workload model consisting of one parameter, cpu think time, would be able to model throughput within 95% accuracy, even without doing any I/O! The only application benchmarks which spend much of their time in I/O are TPC-B and Sort.

Table 3.2 compares Willy's performance on a DECstation 5000 with the performance of the applications being modeled. We see that both Sort and TPC-B can be modeled quite accurately using Willy with appropriate parameter values. Throughput and response time are both accurate within a few percent. This accuracy increases our confidence that the parameters of the synthetic workload capture the first-order performance effects of an I/O workload.

In the next chapter, I develop a benchmark that explores the space of possible workloads, using Willy to measure the performance for each workload.

### 3.7. References

- [Baker91] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff and J. K. Ousterhout, "Measurements of a Distributed File System", *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.
- [Bodnarchuk91] R. R. Bodnarchuk and R. B. Bunt, "A Synthetic Workload Model for a Distributed System File Server", *SIGMETRICS 1991*, May 1991.
- [Chen90] P. M. Chen and D. A. Patterson, "Maximizing Performance in a Striped Disk Array", *Proceedings of the 1990 ACM SIGARCH Conference on Computer Architecture*, Seattle WA, May 1990, 322-331.
- [Coffman72] E. G. Coffman, L. A. Klimko and B. Ryan, "Analysis of Scanning Policies for Reducing Disk Seek Times", *SIAM J. Comput* 1, 3 (September 1972), 269-279.
- [Denning70] P. J. Denning, "Virtual Memory", *ACM Computing Surveys* 2 (September 1970), 153-190.
- [Denning78] P. J. Denning and J. P. Buzen, "The Operational Analysis of Queuing Network Models", *ACM Computing Surveys* 10, 3 (September 1978).
- [Ferrari72] D. Ferrari, "Workload Characterization and Selection in Computer Performance Measurement", *IEEE Computer* 5, 4 (July/August 1972), 18-24.
- [Ferrari84] D. Ferrari, "On the Foundations of Artificial Workload Design", *ACM SIGMETRICS 1984*, 1984, 8-14.
- [Geist87] R. Geist and S. Daniel, "A Continuum of Disk Scheduling Algorithms", *ACM Transactions on Computer Systems* 5, 1 (February 1987), 77-92.
- [Gotlieb73] C. C. Gotlieb and G. H. MacEwen, "Performance of Movable-Head Disk Storage Devices", *Journal of the Association for Computing Machinery (J. ACM)* 20, 4 (October 1973), 604-623.
- [Hill87] M. D. Hill, "Aspects of Cache Memory and Instruction Buffer Performance", UCB/Computer Science Dpt. 87/381, November 1987. Ph.D. dissertation.

- [Hofri80] M. Hofri, "Disk Scheduling: FCFS vs. SSTF Revisited", *Communications of the ACM* 23, 11 (November 1980), 645-653.
- [Jones81] D. A. Jones, "Disk Workload Characterization from Event Trace Analyses", *Proceedings of the 8th International Symposium on Computer Performance Modelling, Measurement and Evaluation*, November 1981, 301-313. Performance 81.
- [Lee91] E. K. Lee and R. H. Katz, "An Analytic Performance Model of Disk Arrays and its Applications", UCB/Computer Science Dpt. 91/660, University of California at Berkeley, 1991.
- [Miller91] E. L. Miller and R. H. Katz, "Input/Output Behavior of Supercomputing Applications", *Proceedings of Supercomputing '91*, 1991, 567-576.
- [Oney75] W. C. Oney, "Queueing Analysis of the Scan Policy for Moving-Head Disks", *Journal of the Association for Computing Machinery (J. ACM)* 22, 3 (July 1975), 397-412.
- [Ousterhout85] J. K. Ousterhout, H. Da Costa and *et al.*, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System", *Operating Systems Review* 19, 5 (December 1985), 15-24. Proceedings of the 10th Symp. on Operating System Principles.
- [Rau79] B. R. Rau, "Program Behavior and the Performance of Interleaved Memories", *IEEE Transactions on Computers* C-28, 3 (March 1979), 191-199.
- [Rodriguez-Rosell76] J. Rodriguez-Rosell, "Empirical Data Reference Behavior in Data Base Systems", *IEEE Computer* 9, 11 (November 1976), 9-13.
- [Rosenblum91] M. Rosenblum, Sprite File Locking, personal communication, October 29, 1991.
- [Smith78] A. J. Smith, "Sequentiality and Prefetching in Database Systems", *ACM Transactions on Database Systems* 3, 3 (1978), 223-247.
- [Smith85] A. J. Smith, "Disk Cache-Miss Ratio Analysis and Design Considerations", *ACM Transactions on Computer Systems* 3, 3 (August 1985), 161-203.

- [Stone72] D. L. Stone and R. Turner, "Disk Throughput Estimation", *Proceedings of the ACM Annual Conference*, August 1972, 704-711.
- [Taylor90] D. Taylor, "Solbourne Filesystem Benchmarks--User Guide", Solbourne technical report, October 29, 1990.
- [Teorey72] T. J. Teorey and T. B. Pinkerton, "A Comparative Analysis of Disk Scheduling Policies", *Communications of the ACM* 15, 3 (March 1972), 177-184.
- [Wilhelm77] N. C. Wilhelm, "A General Model for the Performance of Disk Systems", *Journal of the Association for Computing Machinery (J. ACM)* 24, 1 (January 1977), 14-31.

---

## Chapter 4

# A Self-Scaling Benchmark

---

### 4.1. Overview

Willy is a flexible, general I/O workload generator. But, by itself, it is not the ideal I/O benchmark. This is because regardless of what set of values are assigned to Willy's parameters, it suffers from the classic I/O benchmarking problems: it does not provide understanding into system performance; it does not scale over current or future systems; it does not allow comparisons of systems, except for the measured workload; and each workload is relevant to only a narrow range of applications. Over time, poor scaling renders benchmarks irrelevant—what was an I/O-intensive program this year may not be I/O-intensive next year.

In this chapter, I describe a method that addresses these shortcomings. The result is an evaluation tool that automatically explores the space of possible workloads, running Willy to

measure performance for each workload. In the process of exploring the workload space, the benchmark accomplishes two major goals:

- It shows how performance depends on each parameter and reports which workloads perform well. This increases one's understanding of the system being measured.
- It automatically scales itself by choosing which workloads to report based on each system's performance capabilities. This scaling helps ensure the benchmark's long-term relevance. Unfortunately, scaling also results in running different workloads for different systems, making it difficult to compare two systems directly. Chapter 5 addresses the problem of comparing two systems by using this chapter's results to estimate performance for arbitrary workloads.

This chapter describes how the self-scaling benchmark accomplishes these goals, giving examples and implications along the way. In addition to the workstations benchmarked in Chapter 2, I also measure a Convex C240 mini-supercomputer, a Solbourne SE/905 file server, a raw-I/O interface on the SPARCstation 1+, two client-server configurations (Sun's Network File System and HP's Distributed Unix), and a beta release of an unannounced workstation.

This chapter is divided into two parts. In the first part, Sections 4.3-4.4, I describe my first attempt to create a self-scaling benchmark, using a "knee of the curve" concept to scale all the parameters. This approach, which I call the *Global-Knee Self-Scaling Benchmark*, is quite useful but has several problems. In the second part, Section 4.5, I revise the method and back off from scaling all parameters using the knee of the curve. Instead, I scale a subset of the parameters using the midpoints of each parameter's range.

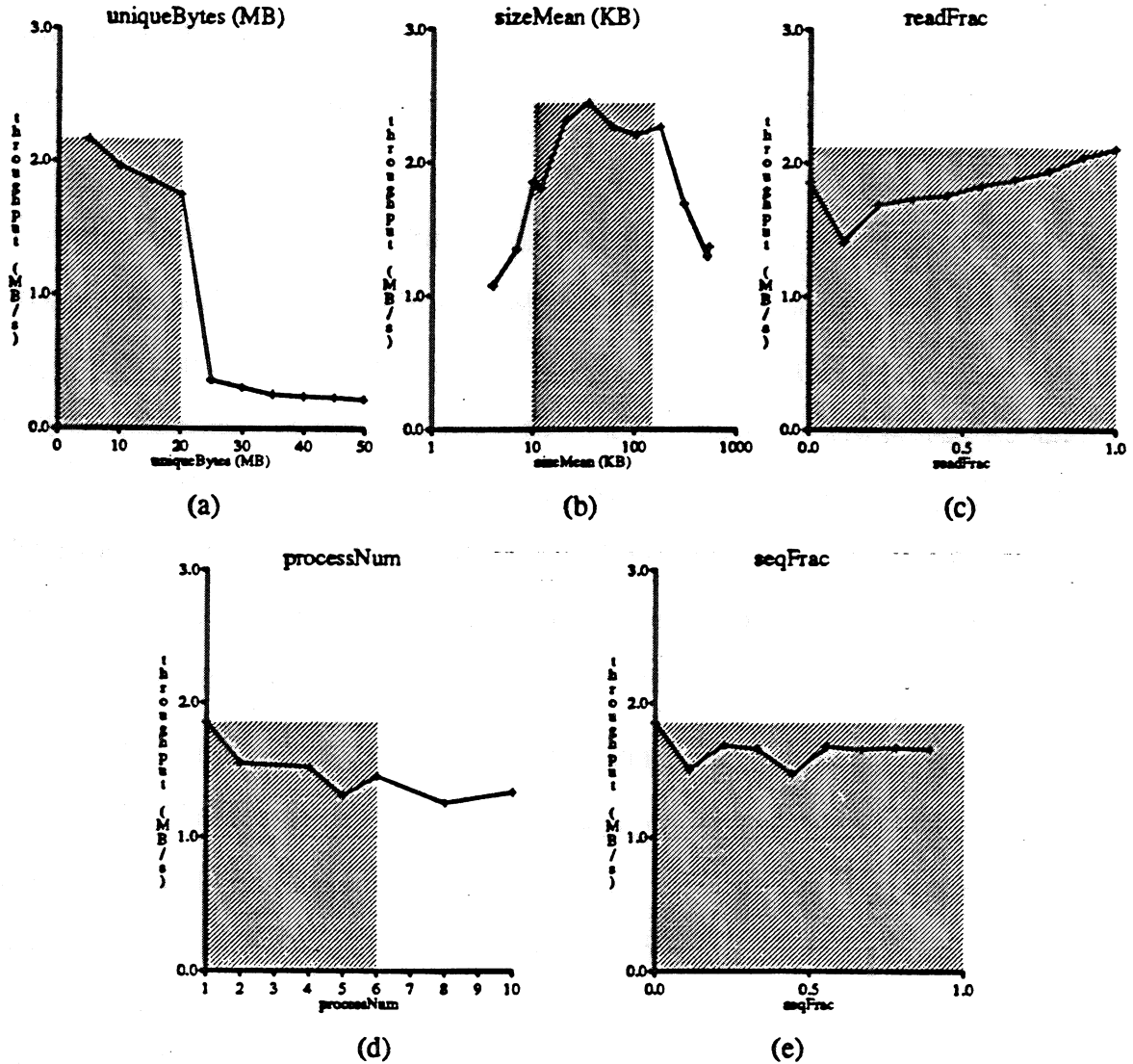
## 4.2. Single Parameter Graphs

Most current benchmarks described in Chapter 2 report the performance for a single workload only. The better benchmarks report performance for multiple workloads, usually in the

form of a graph. TPC-B and Sdet, for example, report how performance varies with load (Figure 2.1 on page 23 and Figure 2.2 on page 25). But even these better benchmarks do not show in general how performance depends on parameters such as request size or the mix of reads and writes.

The main output of Chapter 4's self-scaling benchmark is a set of performance graphs, one for each parameter (uniqueBytes, sizeMean, readFrac, processNum, and seqFrac) as displayed in Figure 4.1. While graphing one parameter, all other parameters remain fixed. I call the value at which a parameter is fixed while graphing other parameters the *focal point* for that parameter. The vector of all focal points is called the *focal vector*. In Figure 4.1, for example, the focal vector is {uniqueBytes = 21 MB, sizeMean = 10 KB, readFrac = 0, processNum = 1, seqFrac = 0}. Hence, in Figure 4.1a, uniqueBytes is varied while the other parameters remain fixed (sizeMean = 10 KB, readFrac = 0, processNum = 1, and seqFrac = 0).

Figure 4.2 illustrates the workloads reported by one set of graphs for a three parameter workload space. Although these graphs show much more of the entire workload space than do current benchmarks, they still show only single parameter performance variations; they do not display dependencies between parameters. Unfortunately, a complete exploration of the entire five dimensional workload space requires far too much time. For example, an orthogonal sampling of six points per dimension requires  $6^5$ , or almost 8000, points. On the Sprite DECstation, each workload takes approximately 10 minutes to measure; thus, 8000 points would take almost 2 months to gather! In contrast, measuring six points for each graph of the five parameters requires only 30 points and 5 hours. The usefulness of these single parameter graphs depends entirely on how accurately they characterize the performance of the entire workload space. Chapter 5 will show that the *shapes* of these performance curves are relatively independent of the specific values of the other parameters. The rest of this chapter discusses how to scale the benchmark by choosing the focal vector.



**Figure 4.1: Results from a self-scaling benchmark that scales all parameters.** This figure shows the results from a self-scaling benchmark of a SPARCstation 1 with 28 MB of memory and a single, SCSI disk. The self-scaling benchmark reports performance for many I/O workloads, where each workload is measured by running Willy with various parameter values. The benchmark reports the range of workloads, shown as the shaded region, that perform well on this system. For example, this SPARCstation system performs well if the total number of unique bytes touched is less than 20 MB. It also shows how performance varies with each workload parameter. Each graph varies exactly one parameter, keeping all other parameters fixed at their focal point. For these graphs, the focal point is the same as the global knee point. The knee point for each parameter is defined to be the least restrictive workload value (see Section 4.3) that yields at least 75% of the maximum performance. The range of workloads that perform well (shaded area), is defined as the range of values that yields at least 75% of the maximum performance. The knee points chosen by the benchmark for each parameter are uniqueBytes = 21 MB, sizeMean = 10 KB, readFrac = 0, processNum = 3, seqFrac = 0.



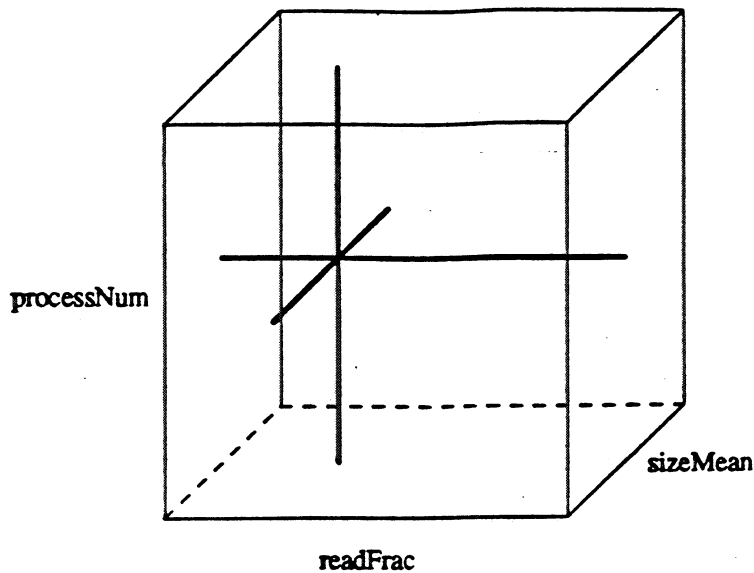


Figure 4.2: Workloads reported by a set of single parameter graphs. This figure illustrates the range of workloads reported by a set of single parameter graphs for a workload of three parameters.

---

### 4.3. The Knee Point

A common concept in engineering is that of the *knee of the curve*, or a *knee point*. This phrase usually refers to an intuitive balance point between higher benefit and higher cost. In the realm of I/O performance, benefit refers to higher performance achieved by different values of workload parameters, while cost refers to the difficulty in creating workloads with those values.

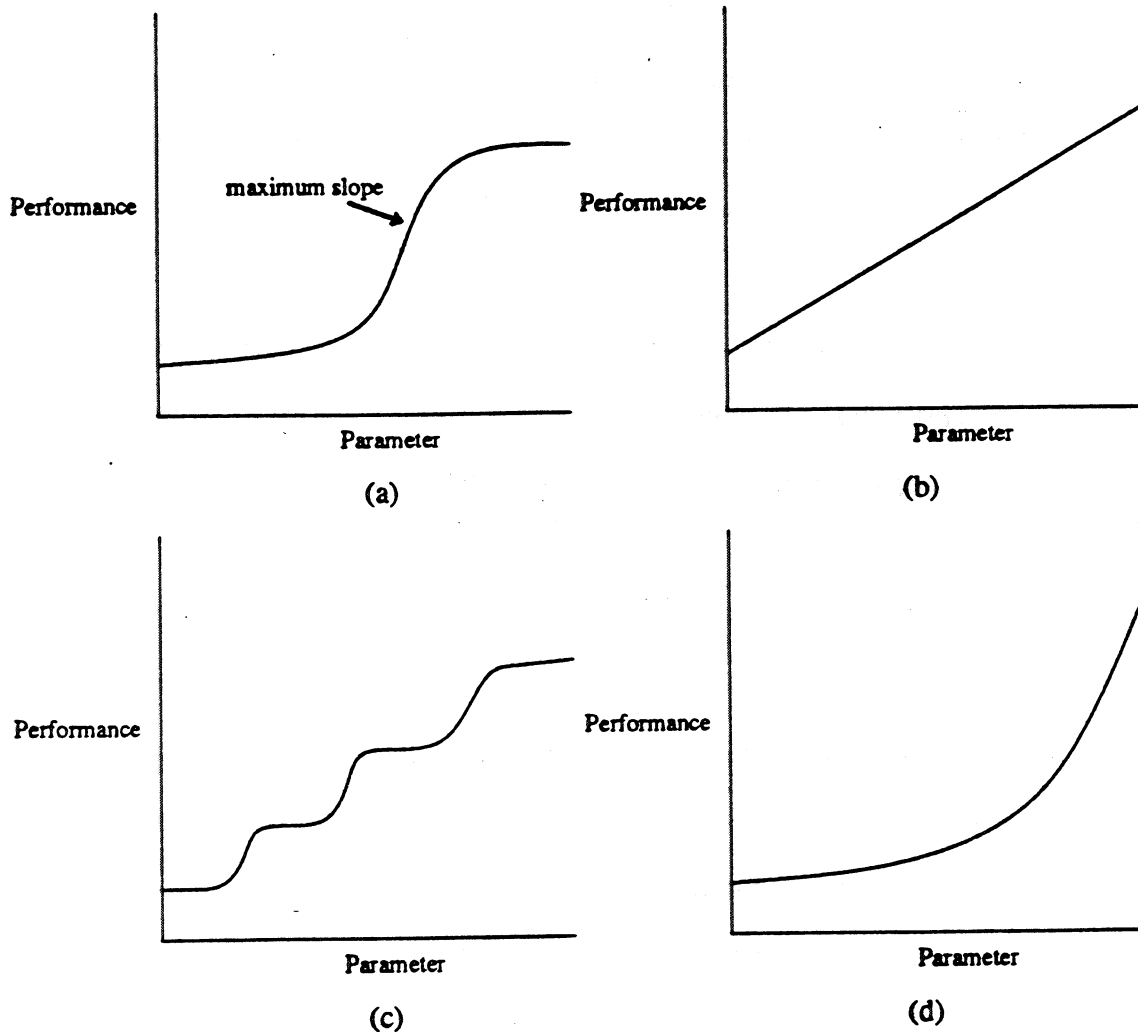
Most of Willy's workload parameters have well-defined directions of higher performance and higher cost. Take `sizeMean` as an example. Larger requests result in higher performance by amortizing any per-request overheads over more bytes. On the other hand, large request sizes are usually difficult to achieve due to practical constraints such as internal fragmentation, long latencies to the last byte, and small file sizes. This tradeoff between practical constraints and higher performance typifies the engineering tradeoffs needed to find a knee point that will provide good performance at reasonable cost. The following is a list of the directions of higher

performance and higher cost for Willy's workload parameters:

- **uniqueBytes**—It is harder to achieve smaller uniqueBytes than larger one. But, smaller uniqueBytes yield higher performance than do larger ones. As a result, programmers often expend much effort to recode their programs in such a way as to work within smaller localities to allow the working set to fit within available memory.
- **sizeMean**—Large sizes are harder than small sizes and leads to higher performance, as explained above.
- **readFrac**—ReadFrac is the only parameter that has no clear direction of higher cost or higher performance. Some scientific applications try to minimize reads, since reads force the user to wait for the data. In contrast, applications such as databases try to minimize writes, since writes must be committed to disk. I assume that whichever direction yields higher performance is also the direction of higher cost.
- **processNum**—Workloads with more processes are harder to achieve than less processes. It is easy to allow only one process at a time to make I/O requests, increasing workload's I/O concurrency is not easy. Increasing the number of processes often increases performance by making use of available parallelism in the disk or memory system. More processes than necessary to achieve maximum throughput is undesirable, however, because response times would degrade without improving throughput.
- **seqFrac**—Higher seqFrac is harder but leads to higher performance. Many applications naturally generate sequential accesses [Ousterhout85], but, in general, it is easier to generate arbitrary access patterns without regard to sequentiality.

For all parameters, except perhaps readFrac, the direction of higher difficulty is also the direction of higher performance. Murphy's Law strikes again!

It is very difficult to derive a mathematically satisfying definition of the knee of a curve. When I started this research, I expected to find a formula, related to the first derivative of the



**Figure 4.3: Difficulty in defining the knee.** Differently shaped performance curves make it difficult to define a universal formula for the knees of curves.

performance, that would define a unique point at which the marginal benefits decreased. Unfortunately, no such formula exists for arbitrarily-shaped performance curves. For the shape of some curves, the choice is fairly easy. Figure 4.3a shows a curve whose knee can be defined as the point of maximum slope. The choice, however, is much less clear for Figure 4.3b, 4.3c, and 4.3d. One stumbling block to defining the knee in terms of the slope of the performance curve

is the scale of the axes. This is because the scale of most parameter's axes is defined independently from the scale of the performance axis. Choosing a knee according to the value of the slope would hence lead to different knees for different axes scales. In the absence of any aesthetically pleasing mathematical definition, I chose an engineering solution: I define the knee of the curve as the point that reaches 75% of the maximum performance. Most of the shortcomings of this knee approach to scaling, listed in Section 4.4.2, will hold regardless of the actual fraction used.

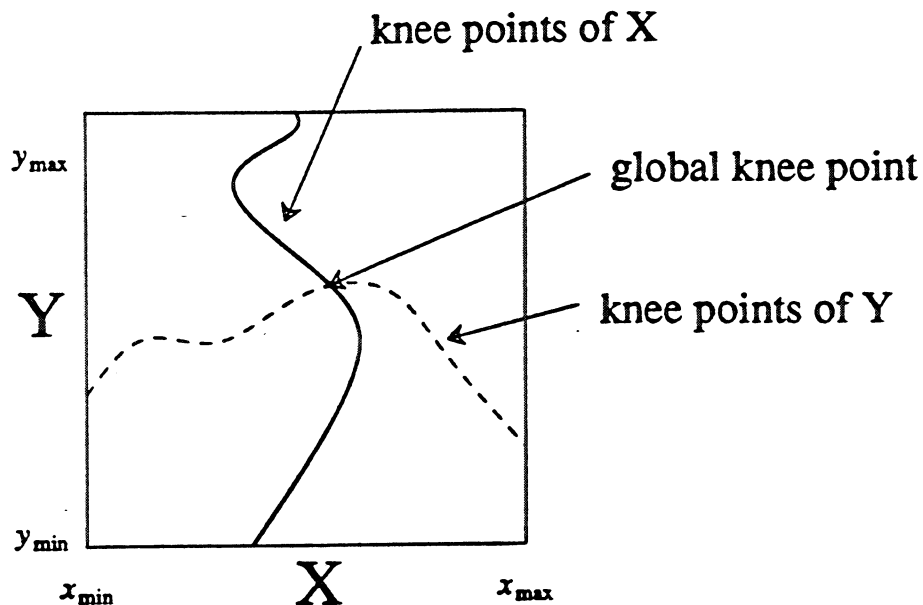
The knee point of Willy's workload parameters provides a hint of how a system might be used. This positive correlation between a system's knee and the use of a system may occur because people who care about performance will buy a computer system that performs well for their workload or modify their current one to perform better for their workload. A few people may even re-code their application to better utilize their system's capabilities. The following are some examples.

- If a potential buyer of a disk system anticipates many concurrent I/Os, he is more likely to buy and use a disk array, which supports higher concurrency (shown by a larger knee value in the processNum performance graph), than a single-disk system.
- If an important program, or class of programs, is thrashing on the system due to a small file cache, the user of that system will likely buy more memory to allow the program to fit in the cache. The RAID-II project [Lee92], for example, bought more memory for the SPARCstation that routed the board to prevent thrashing.
- Programmers for a small class of applications will be able to recode their programs to take advantage of potential performance improvements in moving toward the knee point. Designers of operating systems can increase the block size of the file system; databases can perform parallel queries to increase concurrency [Hong91]; scientific application writers can recode their matrix reference patterns to stay within the file system cache [Kim87].

For some parameters, users will probably always run workloads with a variety of values on any given system. For instance, they will likely always run applications that both read and write data in varying proportions.

The knee point concept can be extended to multiple parameters. I define *global knee point* to be a point at which all parameters are simultaneously at their knee point. As shown below, at least one global knee must exist, assuming the knee points for each parameter are continuous over the workload space.

For a workload with two parameters, X and Y, I can find all possible knee points for X with Y fixed at  $y$ . If I do this for all values of Y between  $y_{\min}$  and  $y_{\max}$ , I graph a curve, such as the solid line shown in Figure 4.4. Note that because knee points are continuous, this curve is continuous and exists for all points between  $y_{\min}$  and  $y_{\max}$ . The same procedure can be carried



**Figure 4.4: Global knee point with two parameters.** This figure proves the existence of a global knee point with two parameters, X and Y. Assuming knee points are continuous, the knee point curve for each parameter must cross in at least one place.

---

out to find all knee points for Y for X fixed at all values between  $x_{\min}$  and  $x_{\max}$ .

For a workload with three parameters, X, Y, and Z, the same can be shown. I can find the knee point of Z for each combination of (X,Y), which appears graphically as a smooth surface over the entire X-Y plane. Because the surface covers the entire X-Y plane, it must intersect with a similar surface for Y's knee point. Intersecting all three knee point surfaces results in at least one global knee point. The next section describes how to find and use this global knee point to create a self-scaling benchmark.

#### 4.4. A Global Knee Self-Scaling Benchmark

A self-scaling benchmark is one that adjusts the workloads that it runs and compiles a report based on the capabilities of the system being measured. Sdet and TPC-B both do this for one aspect of a workload, that is, load (processNum) [SPEC91, TPCB90]. Sdet reports the maximum throughput, which occurs at different loads for different systems. TPC-B reports maximum throughput subject to a response time constraint; this also occurs at different loads for different systems. In this section, I describe a benchmark that generalizes this scaling concept to all workload parameters.

One way to create a self-scaling benchmark is to set the focal vector to be the same as the global knee point. I call this version of the self-scaling benchmark the *global-knee, self-scaling benchmark*.

The global-knee, self-scaling benchmark uses a simple, iterative approach to finding the global knee point. As it runs, it fixes all but one parameter and finds the knee for that one parameter. It then fixes the value for that parameter at the newly found knee point and continues on to another parameter. This iterative process continues until all parameters are simultaneously at their knee points. Although this process could theoretically never converge, in reality very few iterations are needed (only four were needed in the example below). This quick con-

vergence is due to the shape, and thus the knee, of one parameter's curve being reasonably independent of the other parameters. This independence is used and verified in Chapter 5.

After finding the global knee point, the global-knee, self-scaling benchmark reports two items. First, it reports which ranges of workloads perform well on the system. Second, it returns a graph of performance versus each parameter with the other parameters fixed at their focal points. This set of graphs helps the user to understand the performance of the system by showing how performance varies with each parameter.

#### 4.4.1. Example Run of Global-Knee, Self-Scaling Benchmark

This section presents results from a run of the global-knee, self-scaling benchmark. The system being measured is the one-disk SPARCstation of Table 2.1. In Figure 4.1 on page 52, the shaded region on each graph is the range of workloads that perform well for this system, that is, the workload values that yield at least 75% of the maximum performance. When a parameter is varied, the other parameters are fixed at their global knee points chosen by the benchmark {uniqueBytes = 21 MB, sizeMean = 10 KB, readFrac = 0, processNum = 1, seqFrac = 0}. These are the least restrictive values in the range of workloads that perform well. Some of the conclusions that the global-knee, self-scaling benchmark helps us reach about this system are as follows:

- The effective size of the file cache is 21 MB. Applications that have a locality space larger than 21 MB will go to disk frequently.
- I/O workloads with larger average sizes yield higher throughput than smaller average sizes. To get reasonable throughput, request sizes should be between 10 KB and 200 KB. This information may help programmers of operating systems in choosing the best block size.
- Workloads consisting mostly of reads perform slightly better than workloads with more writes.

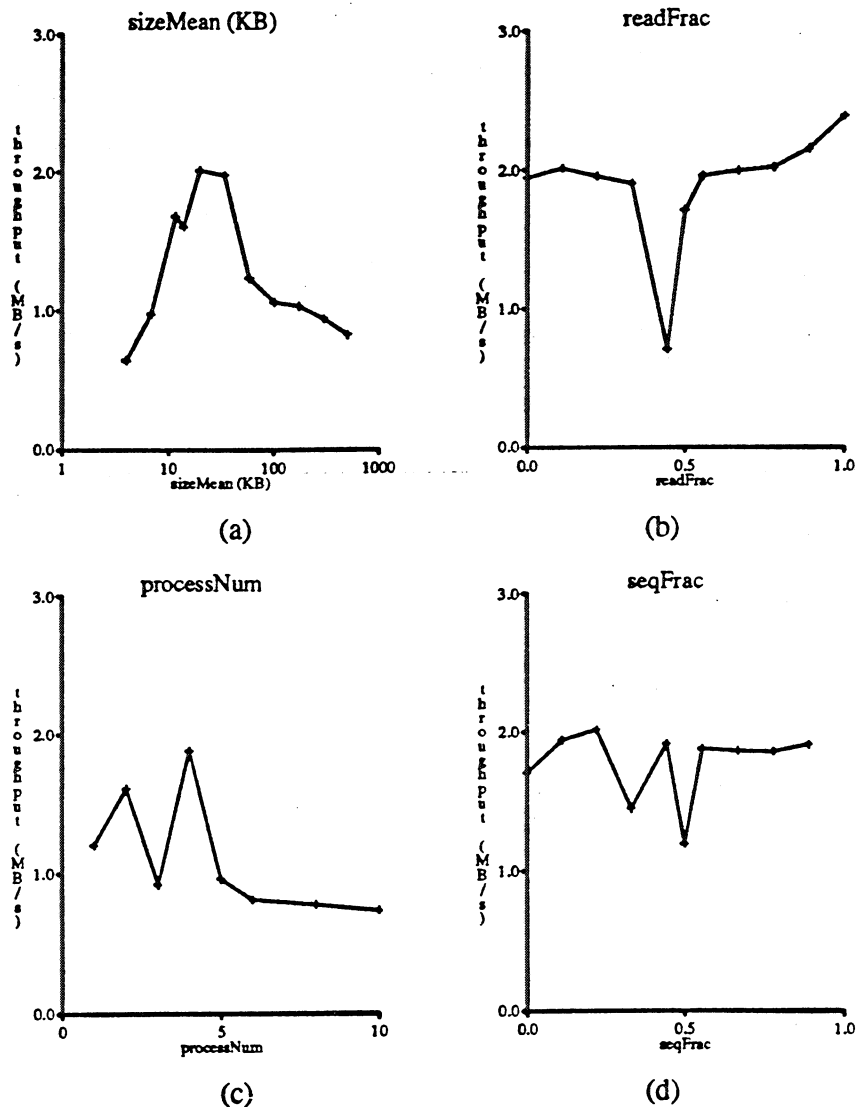
- Increasing concurrency does not improve performance. As expected, without parallelism in the disk system, workload parallelism is of little value.
- Sequentiality does not affect performance at the global knee.

#### 4.4.2. Problems

The global-knee, self-scaling benchmark yields interesting results and helps one understand the system. There are, however, several problems with self-scaling all workload parameters by setting the focal vector to be the same as the global knee. First, the iterative process of finding the global knee may be slow because the number of iterations depends on how truly independent the knee point of one parameter is from the values of the other parameters. Because each iteration takes several hours, more than a few iterations may render the benchmark unusable.

Second, there are really two criteria to consider when choosing the focal vector point for a parameter. Take `readFrac` as an example. The first criterion is what has been mentioned above—scaling `readFrac`'s value to the value for workloads most likely to run on this system. But there is also a second criterion: *the performance curves while `readFrac` is at its focal point should apply to workloads where `readFrac` differs from its focal point.* In other words, the shape of the performance curves at the focal point of `readFrac` should be representative of the shape of the performance curves at other values of `readFrac`. When they are, accurate performance prediction can be done based on these curves. This criterion often precludes choosing an extreme value. For example, if reads were 100 times faster than writes, the global-knee, self-scaling benchmark might pick `readFrac = 1.0` as the focal point. But `readFrac`'s focal point of 1.0 may yield performance graphs for the other parameters that do not apply to workloads with some writes. For `readFrac`, the knee of the performance curve may *not* be a good predictor of what workloads are run on a system; hence the criterion of predicting which workloads will be run is not as important as picking a value representative of workloads in general. It would be





**Figure 4.5: Instability in graphs on the border of performance regions.** This figure demonstrates the instability of choosing the focal point for uniqueBytes close to the border of two performance regions. The focal point chosen for this family of curves is uniqueBytes = 23 MB, sizeMean = 14 KB, readFrac = .5, processNum = 2, seqFrac = .5.

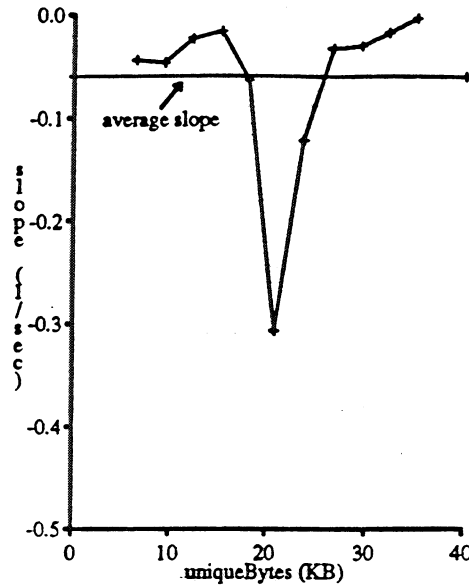
better to choose an intermediary value rather than the knee as the focal point. In fact, as will be seen in Chapter 5, general applicability is a more important criterion than scaling to the knee point, because if the shape of the performance curve is generally applicable, performance can be

estimated for any workload.

The third problem is that the `uniqueBytes` parameter often has distinct performance regions, which correspond to the various storage hierarchies in the system. In Figure 4.1a, `uniqueBytes` smaller than 21 MB primarily use the file cache, while `uniqueBytes` larger than 21 MB primarily use the disk. When the value of `uniqueBytes` is on the border between the file cache and disk region, performance is often unstable (see Figure 4.5, where `uniqueBytes` = 23 MB). The knee point is usually on the border between the storage hierarchy regions, so choosing the focal point for `uniqueBytes` to be the knee point makes it likely that performance will be less stable. Graphing in the middle of a hierarchy level's performance region should be more stable than graphing on the border between performance regions. Another problem that arises from the storage hierarchy regions is that each level of the storage hierarchy may give different performance shapes. Figures 4.8 and 4.9 show how, depending on the storage hierarchy region, reads may be faster than writes (Figure 4.8g), slower than writes (Figure 4.9b), or the same speed as writes (Figure 4.8b).

#### 4.5. A Better Self-Scaling Benchmark

There are a variety of solutions to the problems listed above. For the distinct performance regions discovered by `uniqueBytes`, I measure and report multiple families of graphs, one family for each performance region (Figure 4.1 shows a single family of graphs). These regions are distinguished using a heuristic based on the slope of the `uniqueBytes` curve. For the SPARCstation 1+, the slope of the `uniqueBytes` curve is shown in Figure 4.6. The heuristic delineates performance regions based on what values of `uniqueBytes` are lower than the average slope of the `uniqueBytes` curve. In Figure 4.6, the area near 20 MB is below the average slope, so the heuristic defines two performance regions: one from 0-20 MB, one from 20-60 MB. The self-scaling benchmark measures and reports one family of graphs for each of these performance regions. For instance, the first family of graphs is shown in Figure 4.7a-d and has `uniqueBytes`



**Figure 4.6: Slope of uniqueBytes curve for SPARCstation 1+.** This figure plots the slope of the uniqueBytes curve (Figure 4.7). The one area of the graph with slope below the average slope (0.06 per second) is near 20 MB. The self-scaling benchmark uses this information in a heuristic algorithm to delineate performance regions.

= 12 MB. The second family of graphs, Figure 4.7f-i, has a separate focal point with uniqueBytes = 42 MB.

To improve the general applicability of the graphs, I choose the focal point of each parameter to be in the “middle” of the parameter range. With increasing distance from the focal point, the performance shapes’ applicability will degrade. Using the middle of a parameter’s range minimizes the average overall distance from the focal point and should hence maximize the applicability of the focal point. For parameters such as readFrac and seqFrac, the range is easily defined (0 to 1) hence a midpoint of 0.5.<sup>1</sup> The remaining parameters are uniqueBytes, sizeMean, and processNum; for each performance region, the focal point for uniqueBytes is set at the middle of that region, and for sizeMean and processNum, the focal point is set at the

<sup>1</sup>One could also imagine allowing the focal point to stray from the exact middle of the parameter range toward the knee point for that parameter. This might provide a balance between the two criteria for choosing a focal point. I leave this as possible future research.

value that yields a performance halfway between the minimum and maximum.

After these solutions and modifications are made to the global-knee, self-scaling benchmark, the revised program is called simply the *self-scaling benchmark*.

#### 4.5.1. Examples

This section contains results obtained from running the self-scaling benchmark on five systems—the three systems of Table 2.1, plus a four-processor Convex C240 minisupercomputer and a Solbourne 5E/905 file server, described in Table 4.1.

##### 4.5.1.1. SPARCstation 1+

Figure 4.7 shows the results from the self-scaling benchmark on the SPARCstation 1+. The uniqueBytes values that characterized the two performance regions are 12 MB and 42 MB. Graphs a-d show the file cache performance region, measured with uniqueBytes = 12 MB. Graphs f-i show the disk performance region, measured with uniqueBytes = 42 MB. In addition to what was learned from the first self-scaling benchmark, the following can be seen:

- Larger request sizes yield higher performance. This effect is more pronounced in the disk region.

System Name	Convex C240	Solbourne 5E/905
Year Released	1988	1991
CPU	C2 (4 processors)	SPARC (5 processors)
Speed	220 MIPS	22.8 SPECint
Disk System	4 DKD-502 RAID 5	2 Seagate IPI
I/O Bus	IPI-2	IPI-2
Memory Bus	200 MB/s	128 MB/s
Memory Size	1024 MB	384 MB
Operating System	ConvexOS 10.1 (BSD derived)	SunOS 4.1A.2 (revised)

**Table 4.1: Description of other machines used in self-scaling benchmark examples.** The Convex C240 is a four processor mini-supercomputer. The benchmarked disk system is a four disk RAID 5 with a striping unit of 64 KB. The Solbourne is a network file server for workstations using 2 IPI disks and a stripe unit of 64 KB.

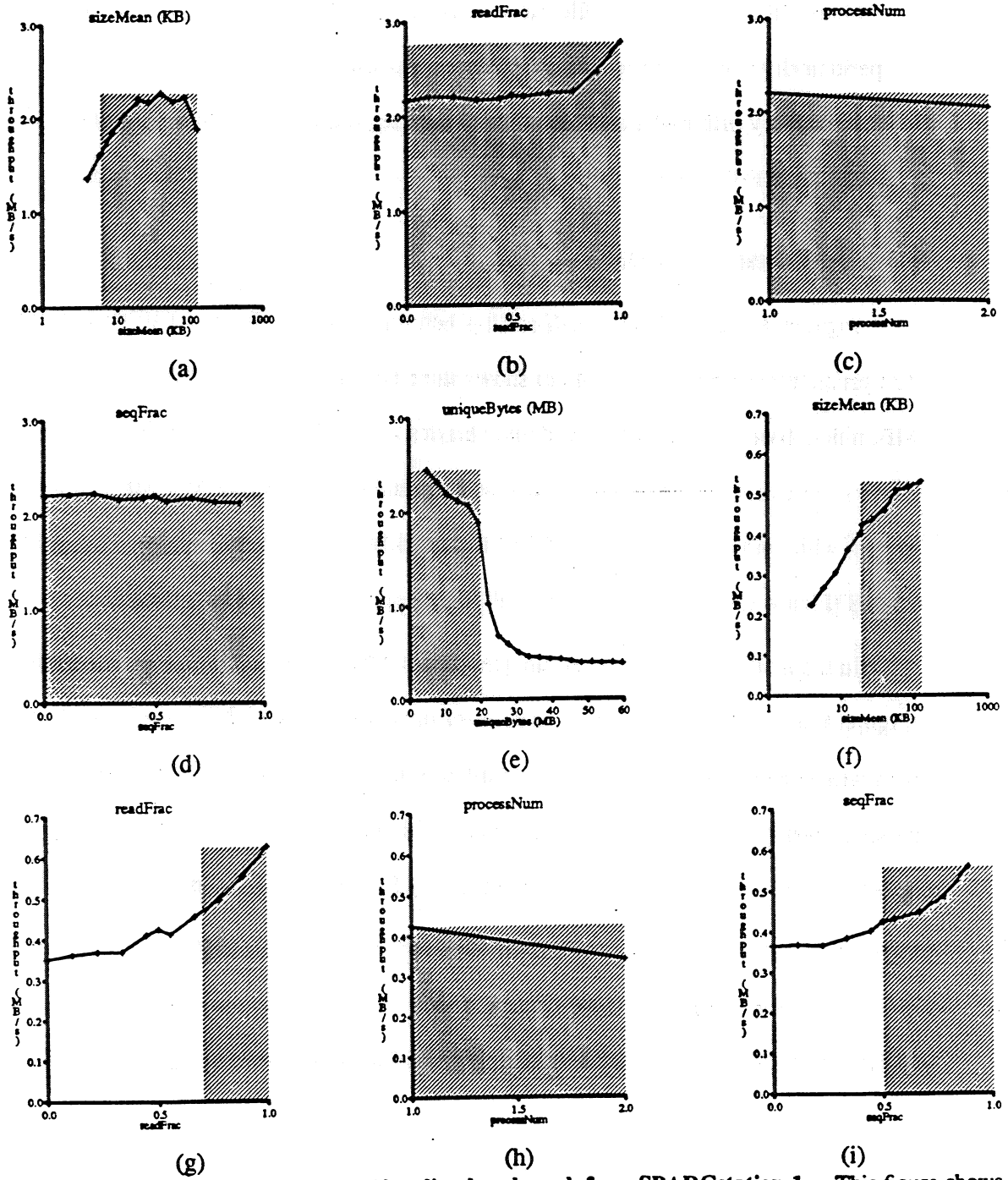


Figure 4.7: Results from a better self-scaling benchmark for a SPARCstation 1+. This figure shows the results from the revised, self-scaling benchmark of the SPARCstation 1+. The focal point for unique-Bytes is 12 MB in graphs a-d and 42 MB in graphs f-i. For all graphs, the focal points for the other parameters is sizeMean = 20 KB, readFrac = 0.5, processNum = 1, seqFrac = 0.5. Increasing sizes improves performance, more so for disk accesses than file cache accesses (Figures a and f). Reads are faster than writes, even when data is in the file cache, because i-nodes must still go to disk. Sequentiality increases performance only for the disk region (Figure i).

- Reads are faster than writes, even when all the data fits in the file cache (Figure 4.1.b). Although the data fits in the file cache, writes still cause i-node changes to be written to disk periodically (for reliability) in case of a system crash.
- Sequentiality offers no benefit in the file cache region (Figure 4.1.b) but offers a substantial benefit in the disk region.

#### 4.5.1.2. DECstation 5000/200

Figures 4.8 and 4.9 show self-scaling benchmark results for the DECstation 5000/200. The uniqueBytes graph (Figure 4.8e) shows three performance plateaus, uniqueBytes = 0 to 5 MB, uniqueBytes = 5 to 20 MB, and uniqueBytes > 20 MB. Thus, the self-scaling benchmark gathers three sets of measurements, one set with uniqueBytes equal to 2 MB (Figure 4.8a-d), one set with uniqueBytes equal to 15 MB (Figure 4.8e-i), and one set with uniqueBytes equal to 36 MB (Figure 4.9). One interesting result is a phenomenon involving readFrac.

In the first performance level (uniqueBytes = 2 MB), reads and writes are the same speed (Figure 4.8b). At the next performance level (uniqueBytes = 15 MB), reads are much faster than writes (Figure 4.8g). This is due to the effective write cache of Sprite's LFS being much smaller than the reads cache, so reads are cached in this range of uniqueBytes while writes are not. The effective file cache size for writes is only 5-8 MB, while for reads it is 20 MB [Rosenblum92].<sup>2</sup> In contrast, when uniqueBytes is large enough to exercise the disk for both reads and writes, writes are faster than reads (Figure 4.9b). This phenomenon is due to Sprite's LFS, which improves write performance by grouping multiple, small writes into fewer, large ones.

---

<sup>2</sup>LFS limits the number of dirty cache blocks to avoid deadlock during cleaning. The default limit was tuned for a machine with 128 MB of memory; in production use, this limit would be changed to match the 32 MB system being tested.

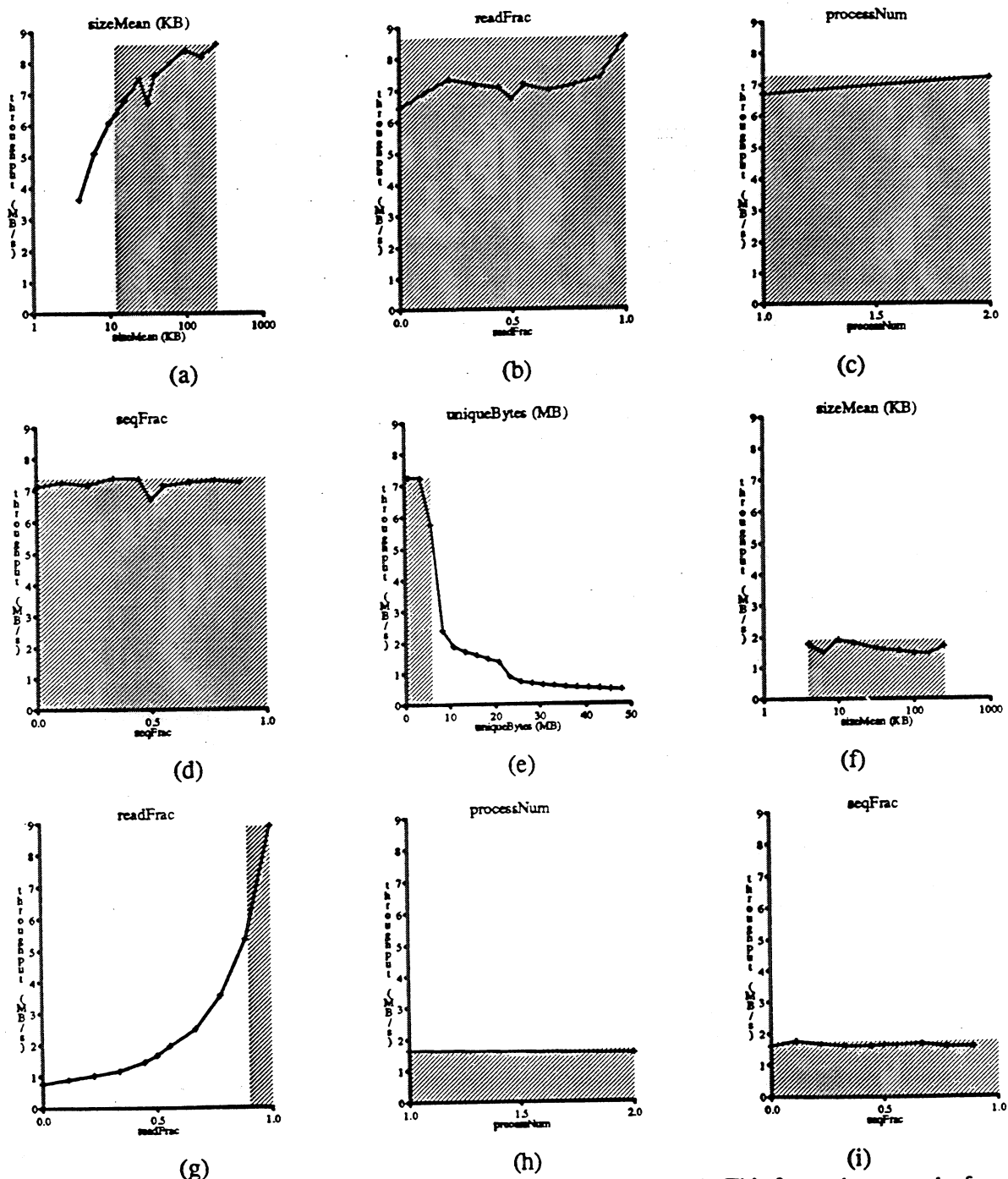
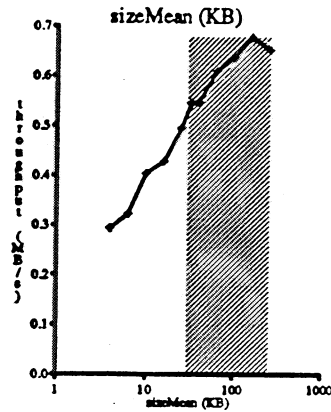
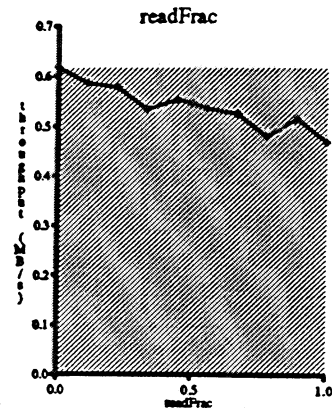


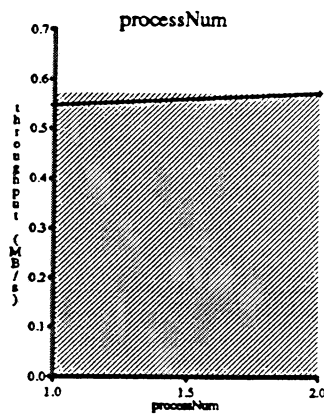
Figure 4.8: Self-scaling benchmark for DECstation 5000/200—Part 1. This figure shows results from the revised, self-scaling benchmark of the DECstation 5000/200. Graph (e) shows three plateaus in uniqueBytes, due to the different effective file cache sizes for reads and writes. Graphs for the third plateau are shown in Figure 4.9. The focal point for uniqueBytes is 2 MB in graphs a-d and 15 MB in graphs f-i. For all graphs, the focal points for the other parameters is sizeMean = 40 KB, readFrac = 0.5, processNum = 1, seqFrac = 0.5. Note how reads can be faster than writes (g), slower than writes (Figure 4.9.b) or the same speed (b).



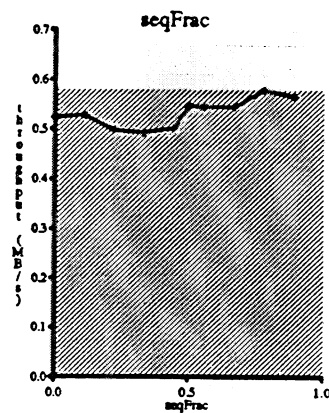
(a)



(b)



(c)



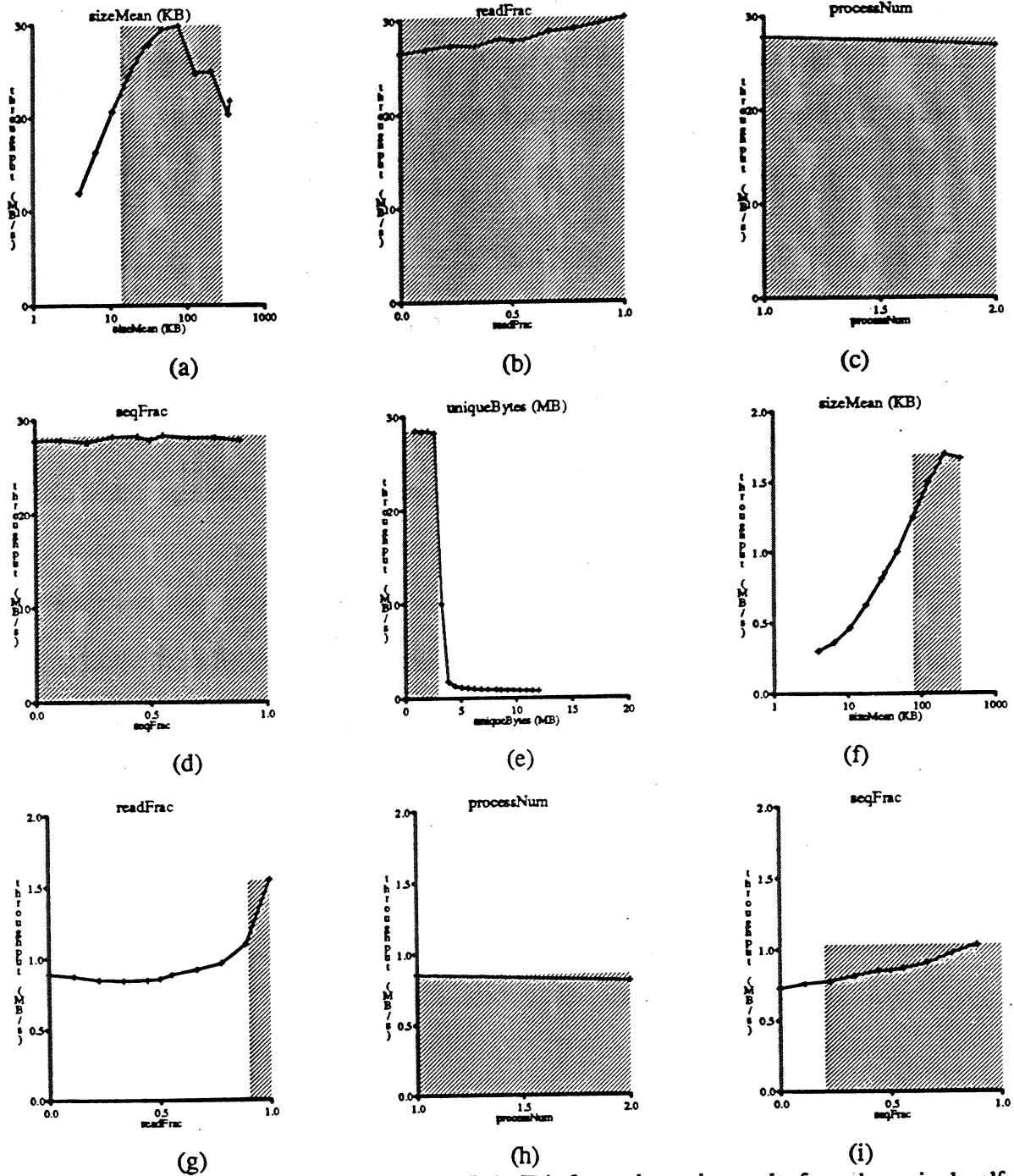
(d)

Figure 4.9: Self-scaling benchmark for DECstation 5000/200—Part 2. This figure continues the results from the revised, self-scaling benchmark of the DECstation 5000/200. The focal point for these graphs is uniqueBytes = 36 MB, sizeMean = 40 KB, readFrac = .5, processNum = 1, seqFrac = .5.

#### 4.5.1.3. HP 730

Figure 4.10 shows the results of the self-scaling benchmark for the HP 730. Note how small the effective file cache is. This is due to the HP/UX operating system's choice not to make all memory available to the file cache. In contrast, SunOS maps files into virtual memory, which allows the file cache to fill the entire physical memory. HP/UX, on the other hand, reserves a fixed amount of space, usually 10%, as the file cache. Since this system has 32 MB of main memory, the file cache is approximately 3 MB. The self-scaling benchmark thus uses





**Figure 4.10: Self-scaling benchmark for HP 730.** This figure shows the results from the revised, self-scaling benchmark of the HP 730. The focal point for uniqueBytes is 1.8 MB in graphs a-d and 8 MB in graphs f-i. For all graphs, the focal points for the other parameters are sizeMean = 32 KB, readFrac = 0.5, processNum = 1, seqFrac = 0.5. Graph (e) reveals that the file cache is much smaller than the size of main memory would usually indicate (3 MB instead of 32 MB). Also note the extremely high performance, up to 30 MB/s, when uniqueBytes is small enough to fit in the file cache.

two focal points, uniqueBytes = 2 MB and uniqueBytes = 6 MB. Also note the high throughput of the HP 730 when accessing the file cache, peaking at almost 30 MB/s for large accesses (Figure 4.10a). This high performance is due to its fast, interleaved memory system (peak memory bandwidth is 264 MB/s) and to the use of a VLSI memory controller that accelerates cache-memory write backs [Horning91].

#### 4.5.1.4. Convex

Results from the self-scaling benchmark of a Convex C240 are shown in Figure 4.11. The curves are similar to those measured on the SPARCstation 1+, but with three main differences:

- Absolute performance is very high. File cache performance reaches 25 MB/s (Figure 4.11b), while disk performance reaches almost 10 MB/s (Figure 4.11f). This high performance is due to Convex's 200 MB/s memory system and performance-focused (as opposed to cost-performance) implementation. Because the Convex disk system is very fast, the performance difference between the file cache and disk region is smaller than for the other systems.
- The effective file cache for the Convex is 800 MB. This is due to the 1 GB of main memory resident on the computer and an operating system that gives the file cache use of the entire main memory.
- Disk and file cache performance continues to improve with increasing size until requests are 2 MB (Figure 4.11f), while most other computers reach their peak performance with sizes of a few hundred kilobytes.

#### 4.5.1.5. Solbourne

Figure 4.12 shows the self-scaling, benchmark results for the Solbourne 5E/905. Two differences from the other graphs are evident.

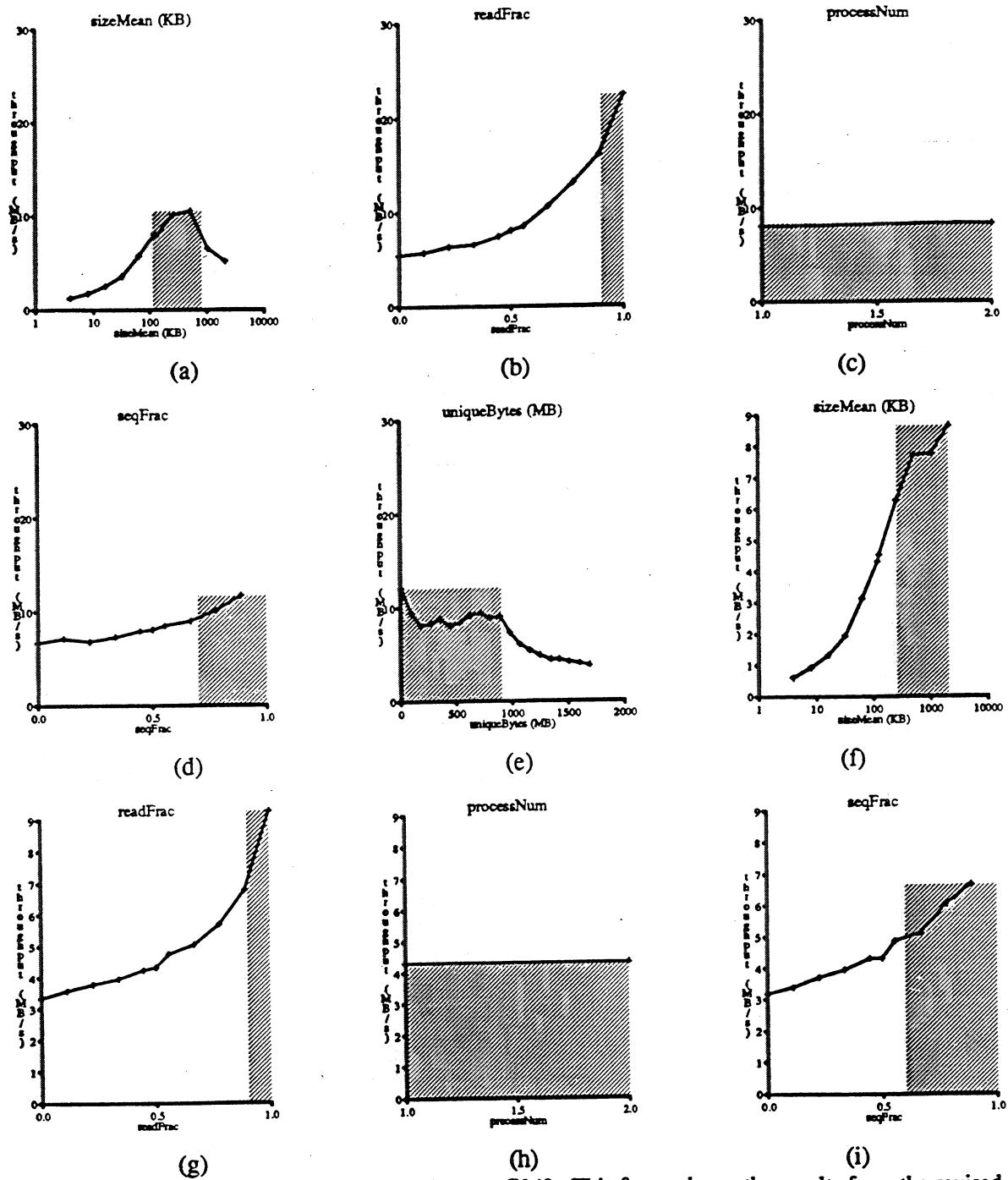


Figure 4.11: Self-scaling benchmark for Convex C240. This figure shows the results from the revised, self-scaling benchmark of the Convex C240. The focal point for uniqueBytes is 450 MB in graphs a-d and 1376 MB in graphs f-i. For all graphs, the focal points for the other parameters are sizeMean = 120 KB, readFrac = 0.5, processNum = 1, seqFrac = 0.5. Note how large the file cache is (800 MB), reflecting how much main memory the system has (1 GB). Also note how large sizeMean grows before reaching its maximum performance (graphs a and f). Because the Convex disk system is very fast, the performance difference between the file cache and disk region is smaller than for the other systems.

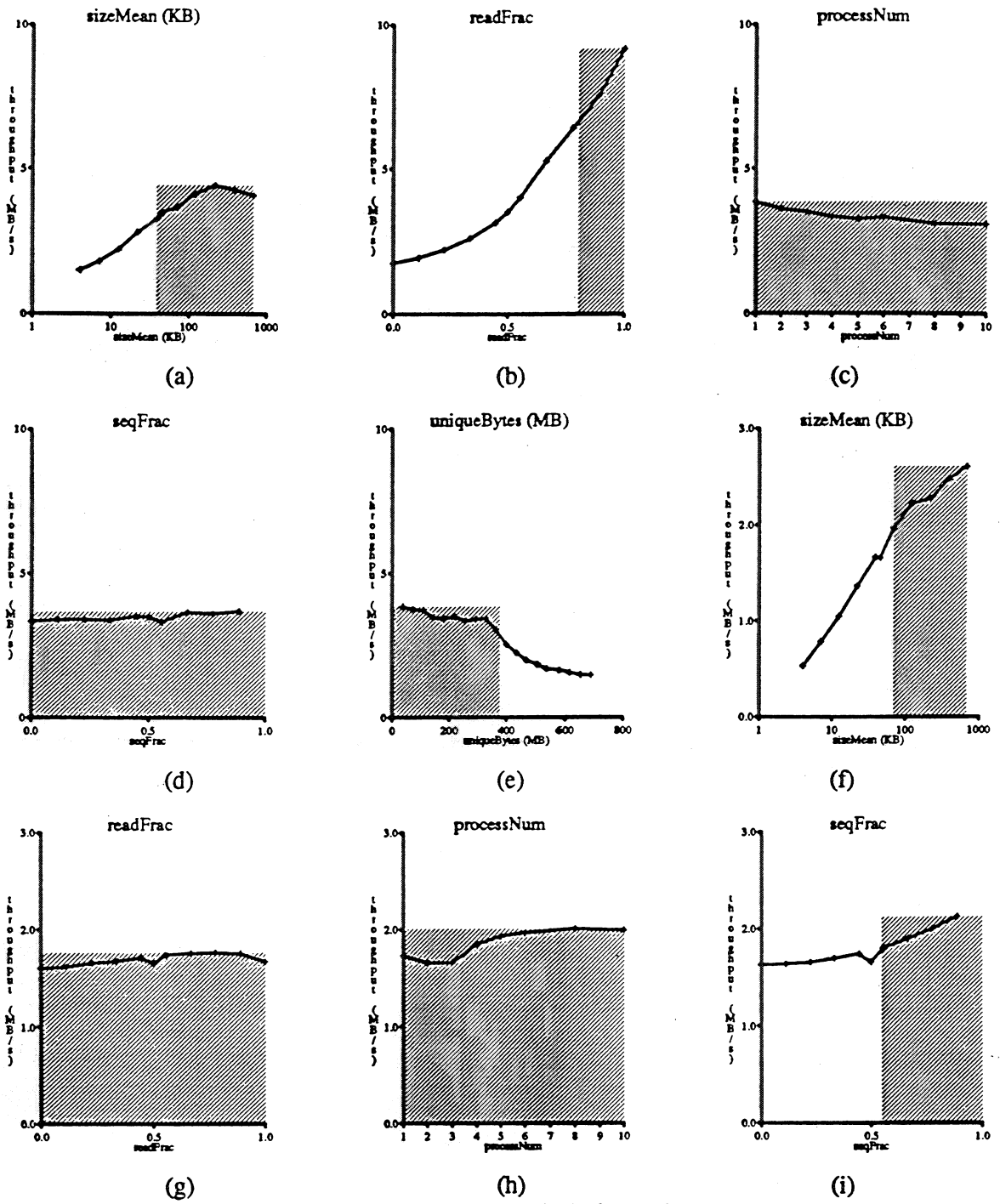
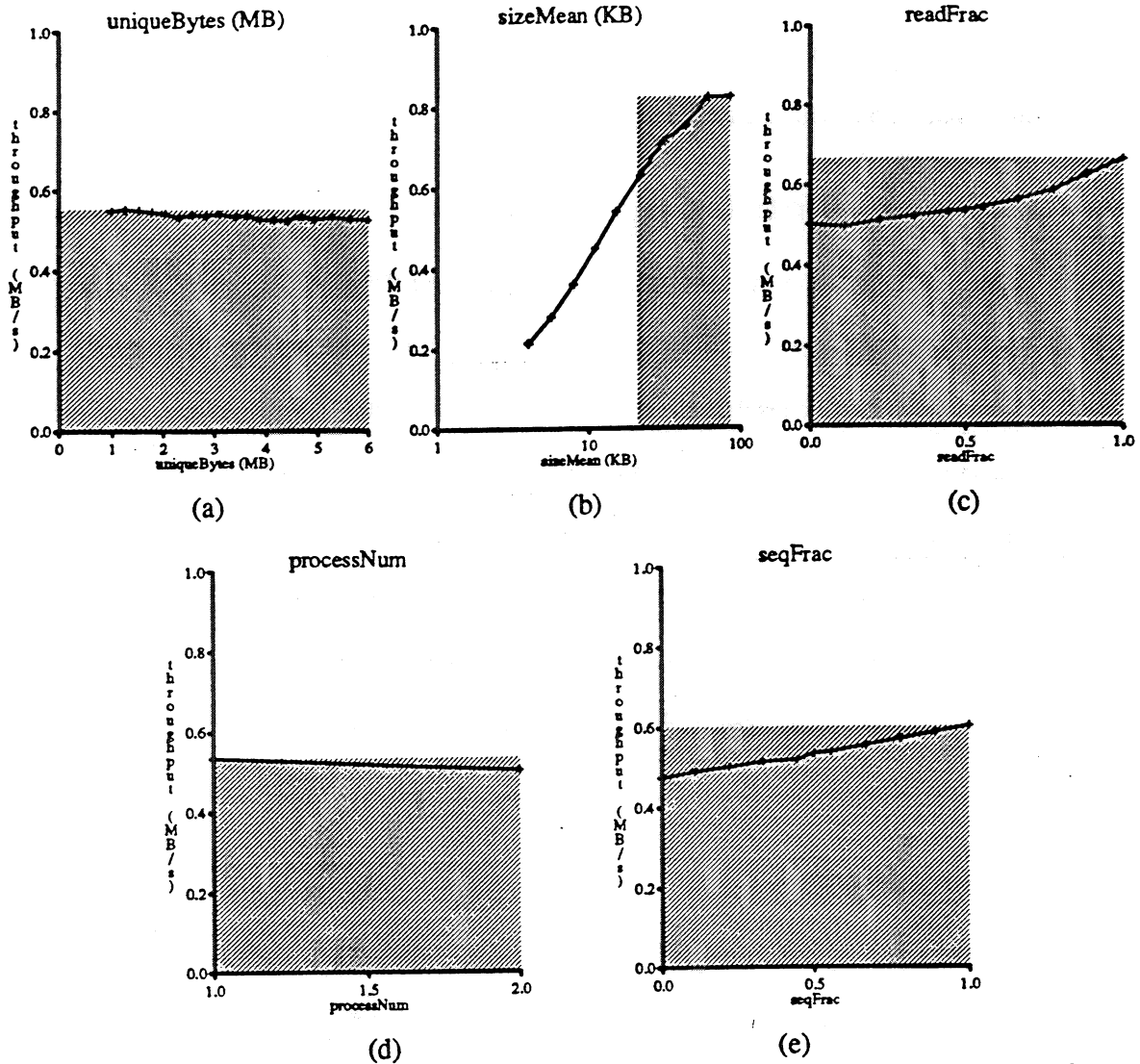


Figure 4.12: Self-scaling benchmark for Solbourne. This figure shows the results from the revised, self-scaling benchmark of the Solbourne. The focal point for uniqueBytes is 180 MB in graphs a-d and 540 MB in graphs f-i. For all graphs, the focal points for the other parameters are sizeMean = 45 KB, readFrac = 0.5, processNum = 1, seqFrac = 0.5. Note how much slower writes are than reads when the file cache is used (graph b). Apparently, the cache write-back policy on the Solbourne causes writes to perform at disk speeds even when all data fits in the file cache.



**Figure 4.13: Self-Scaling Results using the Raw Disk Interface.** This figure shows the results from a self-scaling benchmark of a SPARCstation 1+ using the raw disk interface. The focal point chosen for this interface by the self-scaling benchmark is uniqueBytes 3.5 MB, sizeMean = 15 KB, readFrac = 0.5, processNum = 1, seqFrac = 0.5. The file cache is bypassed when performing I/O through the raw disk interface, hence the uniqueBytes curve is flat in graph (a). Reads are still faster than writes due to the pre-fetching track buffer on the CDC disk which benefits reads more than writes.

- The file cache is quite large, about 300 MB (Figure 4.12e), which is consistent with a main memory size of 384 MB.
- When accessing the file cache, writes are drastically slower than reads (Figure 4.12b). It appears that the Solbourne file cache uses a writing policy, possibly write-through, that

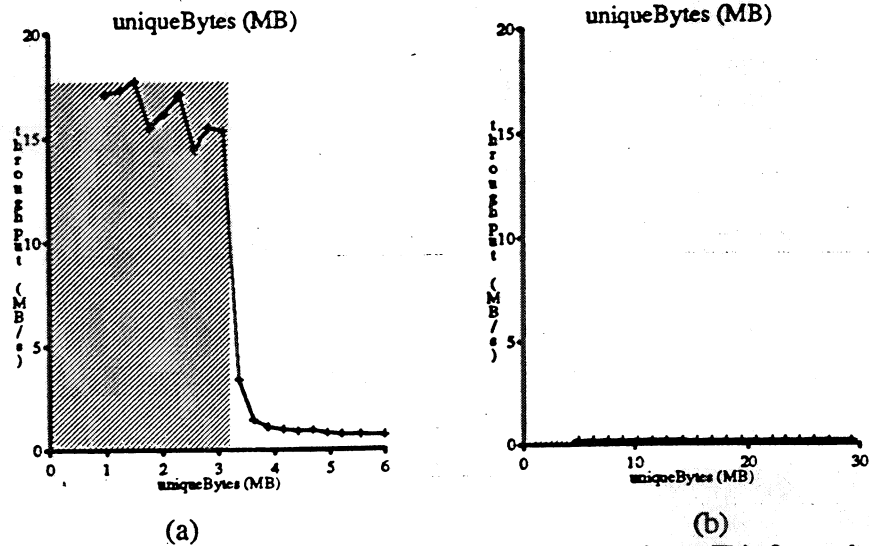
causes writes to the file cache to perform at disk speeds. Because writes derive essentially no benefit from the file cache, performance when varying uniqueBytes changes more gradually than it does with the other systems.

#### 4.5.1.6. Raw Disk Interface

Some applications, such as databases, avoid going through the normal file system interface. Instead, they access the disk directly through the *raw disk interface* [Leffler89]. Figure 4.13 shows benchmark results for the SPARCstation 1+ using the raw disk interface (all other runs have been done using the file system, or *cooked* interface). The main difference of the raw disk interface is its bypassing of the file cache. Hence, the uniqueBytes curve is flat in graph (a). Reads are still faster than writes due to the prefetching track buffer on the CDC disk which benefits reads more than writes. Because the raw disk interface does not go through the file cache, sequentiality improves performance, even at the small uniqueBytes focal point of 3.5 MB.

#### 4.5.1.7. Client-Server

To this point, the self-scaling benchmark has measured the I/O performance of file servers, that is, computers with disks directly attached to them. Many environments instead have client systems connected to a file server via a network. This section gives preliminary results from running the self-scaling benchmark on two different client-server configurations (Figure 4.14). The configuration used in Figure 4.14a is an HP 720 client workstation accessing an HP 730 (same as Table 2.1) file server running DUX (HP's Distributed Unix protocol). The configuration used in Figure 4.14b is a SPARCstation 1+ client workstation accessing a separate SPARCstation 1+ file server running Sun's NFS protocol. Both configurations use an ethernet network to connect the client and server. The main reason the HP configuration gets much higher performance is that the DUX protocol allows clients to cache both reads and writes,



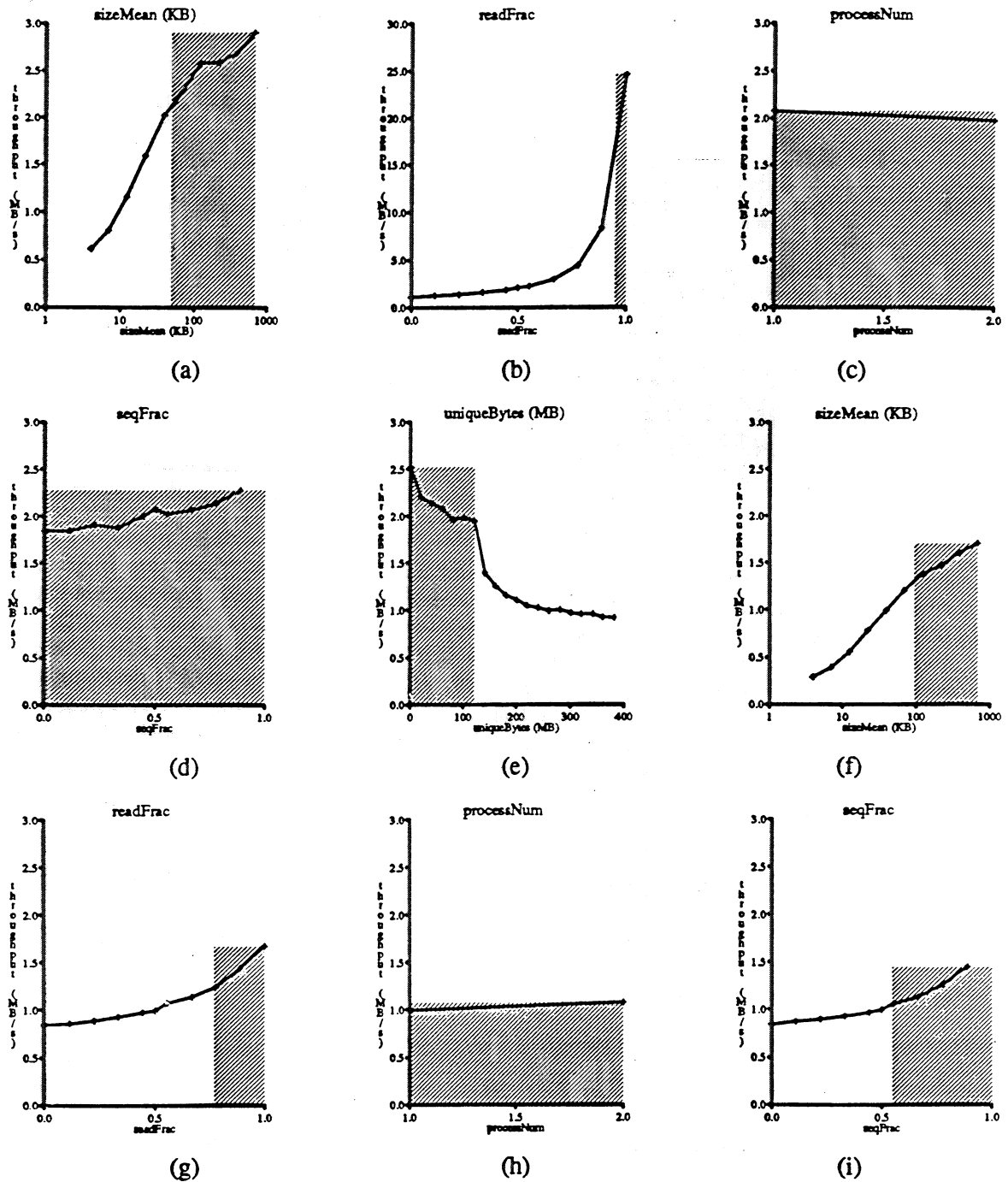
**Figure 4.14: Self-Scaling Results for Two Client-Server Configurations.** This figure shows the results from a self-scaling benchmark of two client-server configurations. The configuration used in graph (a) is an HP 720 client workstation accessing an HP 730 (same as Table 2.1) file server running DUX (HP's Distributed Unix protocol). The configuration used in graph (b) is a SPARCstation 1+ client workstation accessing a separate SPARCstation 1+ file server running Sun's NFS protocol. Both configurations use an ethernet network to connect the client and server. The main reason the HP configuration gets much higher performance is that the DUX protocol allows clients to cache both reads and writes, allowing I/Os to proceed at memory speeds rather than disk or network speeds.

allowing I/Os to proceed at memory speeds rather than disk or network speeds. The disadvantage of the DUX protocol is that reliability suffers because written data stays on the client for 30 seconds before stored on the server.

#### 4.5.1.8. Unannounced Workstation

Figure 4.15 shows preliminary results from running the self-scaling benchmark on a beta release test version of an unannounced workstation<sup>3</sup>. Figure 4.15 reveals several unexpected pieces of information. First, Figure 4.15b shows that, in the file cache region, performance of

<sup>3</sup>This system was running a untuned, beta version of its operating system on pre-production hardware, so performance will likely increase dramatically over the next few months.



**Figure 4.15: Self-scaling benchmark for unannounced workstation (beta release).** This figure shows the results from the revised, self-scaling benchmark of a beta release of an unannounced workstation. The focal point for uniqueBytes is 61 MB in graphs a-d and 261 MB in graphs f-i. For all graphs, the focal points for the other parameters are sizeMean = 40 KB, readFrac = 0.5, processNum = 1, seqFrac = 0.5. Note how much slower writes are than reads when the file cache is used (graph b). Apparently, the cache write-back policy causes writes to perform at disk speeds even when all data fits in the file cache. Note that the scale of the y-axis is a factor of 10 larger in graph (b).



reads is an order of magnitude faster than writes. Like the Solbourne, writes are apparently being written-through the cache directly onto disk. Second, because of the write-through behavior, workloads with any writes get very disappointing performance. Overall system performance is no better than the last generation of workstations. Even with 100% reads, performance is no better than the HP 730. Unlike HP/UX, however, this operating system allows the file cache full use of the 192 MB of main memory.

#### 4.5.2. Running Time

One disadvantage of the self-scaling benchmark is its running time. This is because each performance graph requires approximately 10 points with which to trace a fairly smooth curve. Moreover, with two distinct performance regions, the self-scaling benchmark reports 9 graphs, totaling 90 points. Each point can take 5-10 minutes to measure, due to multiple measurements for tight confidence intervals [MacDougall], which means that the whole process can take half a day or more. On the plus side, the self-scaling benchmark requires no human input and helps the evaluator gain much more insight into system behavior than do traditional, single point benchmarks.

As the capacity of main memory continues to quadruple every 2-3 years [Moore75], file caches will continue to grow larger; hence, larger values of uniqueBytes will be necessary to fill the file cache and stress the underlying disk system. As uniqueBytes increases, the running time of I/O benchmarks that try to stress the disks system will increase proportionally. I can see no way to avoid this linear increase in the running time of any I/O benchmark that goes through the file cache, since each workload measured by Willy will experience this slowdown.

Real applications will, of course, benefit from the trend toward larger memories by needing to go to disk less and less frequently. When I/O benchmarks that fill the cache take too long to run, real applications will take even longer to fill the cache and users will either not care about I/O performance in the steady state or will be willing to use long-running I/O

benchmarks.

#### 4.6. Conclusions

I have shown how a self-scaling benchmark can help an evaluator understand I/O system performance. For example, the evaluations in this chapter provide information on appropriate block sizes, effective file cache sizes for reads and writes, the usefulness of higher concurrency, cache write policies, and absolute throughput. The self-scaling evaluation also met many of the other goals detailed in Section 2.2. It provides an effective, scaling strategy by scaling aspects of the workload to the changing performance characteristics of systems, projecting that users will tend to purchase systems that perform well on their applications. The self-scaling benchmark can be especially useful to the systems programmer trying to understand the performance implications of the I/O code that he or she is writing. For instance, the programmer can develop the code, run the self-scaling benchmark overnight, then see the range of workloads on which the code performs well.

As intended, performance of the benchmark is I/O limited—almost all workloads run here spent more than 90% of their time performing I/O. The self-scaling benchmark is also applicable to a wide range of audiences—by graphing performance curves for each parameter, more applications have their workload included in the benchmark.

However, the self-scaling benchmark does not yet completely meet the goal of meeting the performance evaluation needs for all audiences; there are still many workloads not included in the results of the self-scaling benchmark. Also, self-scaling benchmarks make it difficult to compare two machines directly, because the benchmark will select a different focal vector for each machine. The next chapter addresses these shortcomings by predicting performance for any workload.

#### 4.7. References

- [Convex89] *C Series Data Sheet—Input/Output Subsystems*, CONVEX Computer Corporation, 1989.
- [Convex90] *The First Family of Open Supercomputing*, CONVEX Computer Corporation, 1990.
- [Convex91] *C3200*, CONVEX Computer Corporation, 1991.
- [Hong91] W. Hong and M. Stonebraker, "Optimization of Parallel Query Execution Plans in XPRS", *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, Miami, FL, December 1991. also to appear in *Journal of Distributed and Parallel Databases*.
- [Horning91] R. Horning, L. Johnson, L. Thayer, D. Li, V. Meier, C. Dowdell and D. Roberts, "System Design for a Low Cost PA-RISC Desktop Workstation", *Proceedures of the IEEE Computer Society International Conference (COMPCON)*, Spring 1991, 208-213.
- [IDC91] *Integrated Disk Channel (IDC)*, CONVEX Computer Corporation, 1991.
- [Kim87] M. Y. Kim, A. Nigam, G. Paul and R. J. Flynn, "Disk Interleaving and Very Large Fast Fourier Transforms", *International Journal of Supercomputer Applications* 1, 3 (Fall 1987), 75-96.
- [Lee92] E. K. Lee, P. M. Chen, J. H. Hartman, A. L. C. Drapeau, E. L. Miller, R. H. Katz, G. A. Gibson and D. A. Patterson, "RAID-II: A Scalable Storage Architecture for High-Bandwidth Network File Service", UCB/Computer Science Dpt. 92/672, University of California at Berkeley, February 1992.
- [Leffler89] S. J. Leffler, M. K. McKusick, M. J. Karels and J. S. Quarterman, *The Design and Implementation of the 4.3BSD Unix Operating System*, Addison-Wesley Publishing Company, 1989.
- [MacDougall] M. H. MacDougall, *Simulating Computer Systems, Techniques and Tools*, The MIT Press. Computer Systems Series.
- [Moore75] G. E. Moore, "Progress in Digital Integrated Electronics", *IEEE Digest International Electron Devices Meeting*, 1975, 11.

- [Ousterhout85] J. K. Ousterhout, H. Da Costa and *et al.*, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System", *Operating Systems Review* 19, 5 (December 1985), 15-24. Proceedings of the 10th Symp. on Operating System Principles.
- [Rosenblum92] M. Rosenblum, Sprite LFS Write Cache Size, personal communication, July 6, 1992.
- [SPEC91] *SPEC SDM Release 1.0 Manual*, System Performance Evaluation Cooperative, 1991.
- [TPCB90] *TPC Benchmark B Standard Specification*, Transaction Processing Performance Council, August. 23, 1990.

---

## Chapter 5

# Predicted Performance

---

### 5.1. Introduction

The self-scaling benchmark of Chapter 4 automatically scales its workload according to the capability of the system being measured. This automatic scaling keeps the benchmark relevant over a wide range of machines. By graphing the workloads around a few focal points, the benchmark also provides information on what workloads perform well for a system. There are, however, several shortcomings of the self-scaling benchmark. First, by self-scaling, the benchmark complicates the task of comparing results from two systems. This complication arises because the self-scaling benchmark will likely choose different workloads by which to measure each system. Next, though the performance graphs apply to a wider range of applications than do today's I/O benchmarks, they stop short of applying to all workloads. In this

chapter, I show how *predicted performance* solves these problems by using the results from the self-scaling benchmark to accurately estimate the I/O performance for arbitrary workloads, that is, within 10-15% of actually measuring the workload.

This approach is similar in concept to work done by Saavedra-Barrera who predicts CPU performance by measuring the performance for a small set of primitive FORTRAN operations [Saavedra-Barrera89]. The main difference between the performance of FORTRAN programs and I/O workloads is the way performance depends on operations and arguments. A FORTRAN operation, such as adding two floating-point numbers, takes the same amount of time independent of the values of the arguments. To predict the performance for an arbitrary FORTRAN workload (program), only the performance and frequency for each operation in the program need be known. In Saavedra-Barrera's research, the main difficulty was to choose a complete set of primitive FORTRAN operations.

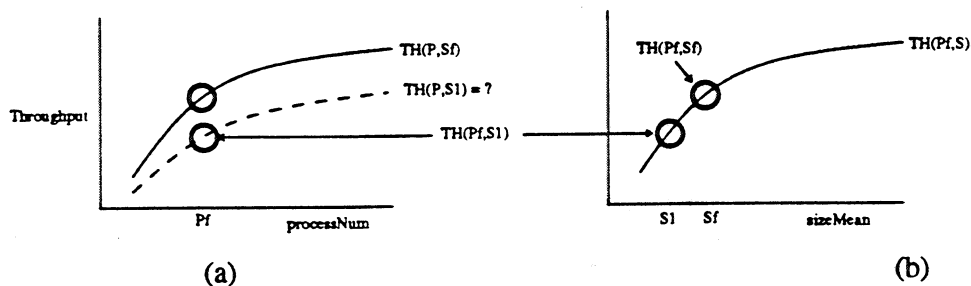
In contrast to FORTRAN programs, I/O has only two operations, read and write, and performance depends heavily on the arguments (sizeMean, uniqueBytes, seqFrac, and processNum). No short list of performances for each operation is possible because such a list would be for each unique combination of uniqueBytes, sizeMean, readFrac, processNum, and seqFrac. Since it is clearly impractical to measure or list all combinations, I develop a method for using the summary performance graphs of Chapter 4 to estimate performance for any combination of workload parameters.

## 5.2. Prediction Models

A straightforward approach to estimating performance for all possible workloads takes a fine-grained, orthogonal sampling of workloads. Performance of arbitrary workloads within this orthogonal mesh could then be estimated by interpolating in multiple dimensions between neighboring points. The main disadvantage is the time it takes to gather the orthogonal sample—the number of points in this sample increases exponentially with the number of

parameters in the workload. Ten sample points for each of the five workload parameters require 100,000 measurements, or two years on the SPARCstation 1+ of Table 2.1 (page 20)! In addition, Section 5.4 will show how, given an equal number of measured workloads, interpolation using orthogonal sampling gives poorer performance prediction than this chapter's proposed method. Instead of interpolation using orthogonal sampling, I take a different approach by using the performance behavior given in the self-scaling benchmark graphs to estimate performance for unmeasured workloads.

Figure 5.1 shows the general approach to estimating performance for unmeasured workloads applied to a two-parameter workload. Intuitively, the simplification that I investigate in



**Figure 5.1: Predicting performance of unmeasured workloads.** This figure shows how to predict performance with a workload of two parameters, processNum and sizeMean. Intuitively, I assume that the *shape* of a performance curve for one parameter is *independent* of the other parameters' values. More specifically, I assume that the *ratio* of performance between two plots against a given parameter (such as sizeMean) is constant. The solid lines represent workloads that have been measured; the dashed line represents workloads that are being predicted. The solid line in Figure 5.1a shows throughput graphed against processNum with sizeMean fixed at sizeMean\_f. Figure 5.1b shows throughput versus sizeMean with processNum fixed at processNum\_f. I predict the throughput curve versus processNum in Figure 5.1a with sizeMean fixed at sizeMean\_1 by assuming that  $\frac{\text{Throughput}(\text{processNum}, \text{sizeMean}_1)}{\text{Throughput}(\text{processNum}_f, \text{sizeMean}_f)}$  is constant (independent of processNum) and fixed at  $\frac{\text{Throughput}(\text{processNum}_f, \text{sizeMean}_1)}{\text{Throughput}(\text{processNum}_f, \text{sizeMean}_f)}$ . For example, if  $\text{Throughput}(\text{processNum}_f, \text{sizeMean}_f)$  is 6 and  $\text{Throughput}(\text{processNum}_f, \text{sizeMean}_1)$  is 3, then I estimate  $\text{Throughput}(\text{processNum}, \text{sizeMean}_1)$  (the dashed line in Figure 5.1a) as  $\frac{3}{6} \times \text{Throughput}(\text{processNum}, \text{sizeMean}_f)$ .

this chapter is the *independence* of the *shape* of one parameter's performance curve from the values of the other parameters. In the next section, I demonstrate that this simplification introduces only a small error. In the self-scaling evaluation, I measure workloads with all but one parameter fixed at a focal point. In Figure 5.1, these are shown as the solid-line throughput curves  $Throughput(processNum, sizeMean_f)$  and  $Throughput(processNum_f, sizeMean)$ , where  $processNum_f$  is processNum's focal point and  $sizeMean_f$  is sizeMean's focal point. Using these measured workloads, I estimate performance for unmeasured workloads  $Throughput(processNum, sizeMean_1)$ , where  $sizeMean_1 \neq sizeMean_f$ , shown as the dashed line in Figure 5.1a. I assume a *constant ratio* between  $Throughput(processNum, sizeMean_f)$  and  $Throughput(processNum, sizeMean_1)$ . This ratio is known at  $processNum = processNum_f$  to be  $\frac{Throughput(processNum_f, sizeMean_f)}{Throughput(processNum_f, sizeMean_1)}$ . My assumption that the performance shapes are independent of other parameter values leads to an overall performance equation of

$$Throughput(X, Y, Z, \dots) = f_X(X) \times f_Y(Y) \times f_Z(Z) \dots$$

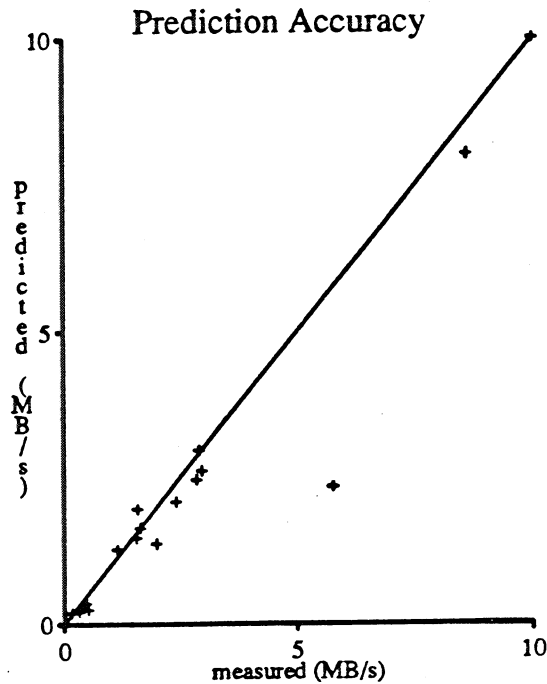
where X, Y, Z, ... are the parameters and  $f_X(X)$ ,  $f_Y(Y)$ , and  $f_Z(Z)$  are the graphs for each parameter (Appendix C proves that if the performance shape for each parameter is independent of the other parameter values, the overall performance equation must take this form).

The above section describes how to estimate performance from a family of performance graphs (Appendix B gives a pseudo-code version). Each performance region has its own set of graphs, as in Figures 4.8 and 4.9 on pages 67 and 68. In estimating performance for a workload, I use the graphs from whichever performance region that workload falls in.

### 5.3. Verification of Prediction Model

In this section, I examine how accurately the equation described above approximates actual performance for the systems measured by the self-scaling benchmark. Ideally, I would like to measure large, I/O-intensive applications and compare their performance to those





**Figure 5.2: Predicting performance of four traditional benchmarks.** This figure graphs the predicted performance against the actual (measured) performance for four traditional benchmarks (Andrew, Bonnie, IOStone, and Sdet) for four systems (SPARCstation 1+, DECstation 5000/200, HP 730, and Convex C240).

predicted using the data collected in the self-scaling benchmarks. Unfortunately, no such public domain applications are available because, as Chapter 2 showed, even well known systems benchmarks are CPU-bound. To assess accuracy given the lack of benchmarks I take two tacks:

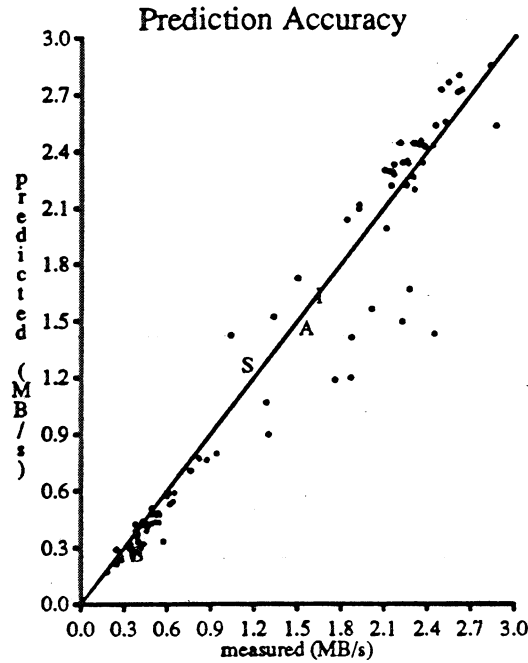
- (1) First, I measure the I/O portion of four I/O benchmarks and compare them to the predicted performance using the self-scaling data for the values of the parameters from the measurement of that I/O benchmark. Figure 5.2 shows a scatter plot of predicted versus measured performance for Andrew, Bonnie, IOStone, and Sdet on four systems: a SPARCstation 1+, a DECstation 5000/200, an HP 730, and a Convex C240. With perfect prediction, predicted performance would exactly equal measured performance, and all points would lie on the 45 degree line. Figure 5.2 shows that this is approximately the case— out of 16 points, all but one are predicted accurately; only Sdet run-

ning on the Convex C240 is substantially in error. These results are typical of the prediction model's accuracy: performance of most workloads is predicted accurately; a few workloads, particularly those far away from the focal point, have significant error. In this experiment, I measured prediction for four workloads designed by independent groups to make sure that prediction is accurate for valued workloads. But there is a danger that these benchmarks do not reflect a wide enough range of current or future workloads.

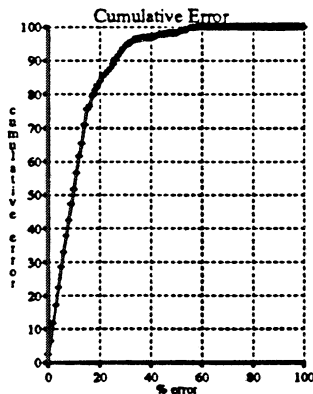
- (2) So, to increase my confidence that the model can predict performance over a wider range of workloads, I measured the performance of 100 random I/O workloads and compare these measurements to their predicted performance. The 100 workloads are randomly selected over the entire space of workloads derived for each system in the self-scaling benchmark. Each workload's predicted performance versus actual, measured performance is then plotted on a scatter plot, as in Figure 5.3. By verifying prediction accuracy over 100 workloads, I became more certain that the average prediction error over these workloads is indicative of the prediction error over the entire workload space. Figure 5.3c shows how the confidence interval from measuring average prediction error tightens with larger numbers of random workloads. By the time I measure 100 workloads, the 90% confidence interval is only a few percent.

### 5.3.1. SPARCstation 1+

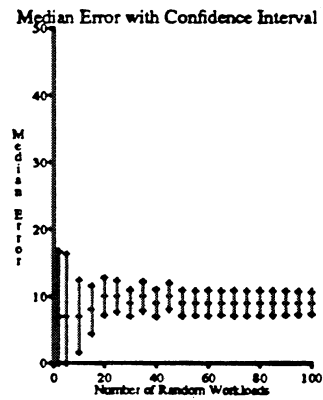
Figure 5.3a shows how accurately the *product of single parameter functions* approach predicts performance for Andrew, Bonnie, IOStone, and Sdet. For each of these benchmarks, the model is able to accurately predict performance. In addition, over the wide range of performances (0.2 MB/s to 3.0 MB/s) resulting from 100 random workloads, the predicted performance values match extremely well to the measured results. Half of all workloads, for example, have a prediction error of 10% or less; three-quarters of all workloads had a prediction error



(a)



(b)



(c)

**Figure 5.3: Evaluation of prediction accuracy for SPARCstation 1+ with one disk.** This figure graphs the predicted performance against the actual (measured) performance for the SPARCstation in Figure 4.7. Points representing traditional benchmarks are shown as the letters “A”, “B”, “I”, and “S”, representing Andrew, Bonnie, IOSTone, and Sdet, respectively. Graph (a) shows how the prediction model accurately predicts performance for these four traditional benchmarks. In addition to predicting performance for a few benchmarks, the model is able to do so for 100 random workloads. Each point represents a single workload, with each parameter value randomly chosen from its entire range shown in Figure 4.7. The closer the points lie to the solid line, the better the prediction accuracy. Median error for graph (a) is 10%; the complete cumulative error distribution is given in graph (b). Performance for each workload ranges from 0.2 MB/s to 3.0 MB/s. Clearly single-point benchmarks cannot adequately capture the large range of performances generated by diverse workloads. Graph (c) gives the 90% confidence interval of the median error, that is, it displays how certain I am that the median error is really 10%. Using 100 workloads, I am 90% sure that the true median error is between 7.5% and 10.7%.

of 15% or less. Figure 5.3b shows the cumulative error distribution of the prediction. In contrast, any single-point I/O benchmark would lead one to believe that all workloads yielded the same performance. For example, both Andrew's and IOStone's workloads yield performances of 1.25 MB/s, leading to a median prediction error of 50%. Bonnie's sequential block write yields a performance of .32 MB/s, for a median prediction error of 65%. Table A.1 in Appendix A gives the raw data used in Figure 5.3.

It is interesting to note where the points of higher error occur, which gives rise to the question: is there a correlation between certain parameters and regions of high error? Figure 5.4 shows how median error varies with each parameter. Note that error is most closely correlated to the value of uniqueBytes. Prediction is particularly poor near the border between performance regions, that is, between the file cache and disk region. As expected, sharp drops in performance lead to unstable throughput and poor prediction.<sup>1</sup> Other than uniqueBytes, prediction accuracy is fairly independent of parameter values.

### 5.3.2. DECstation 5000/200, HP 730, Convex C240

Figures 5.5-5.10 show the prediction accuracy for the Sprite DECstation 5000/200, the HP 730, and the Convex C240 (raw data is given in Appendix A in Tables A.2, A.3, and A.4). For all systems, this chapter's method of predicting performance estimates a wide range of workloads accurately, with median errors ranging from 10-15% (Table 5.1). Table 5.1 also lists the inherent measurement error, which I measured by running the same set of random workloads twice and using one run to "predict" performance of the other run.

Figure 5.6 and 5.8 show how, similar to the SPARCstation 1+, error is usually most closely correlated to uniqueBytes. For the DECstation 5000/200, performance is unstable and

---

<sup>1</sup>If a user's workload fell near the border between performance regions, then instead of trying to accurately predict his workload's performance, his effort would be better spent in buying more memory for his system to expand the file cache region and hence to vastly improve his performance.

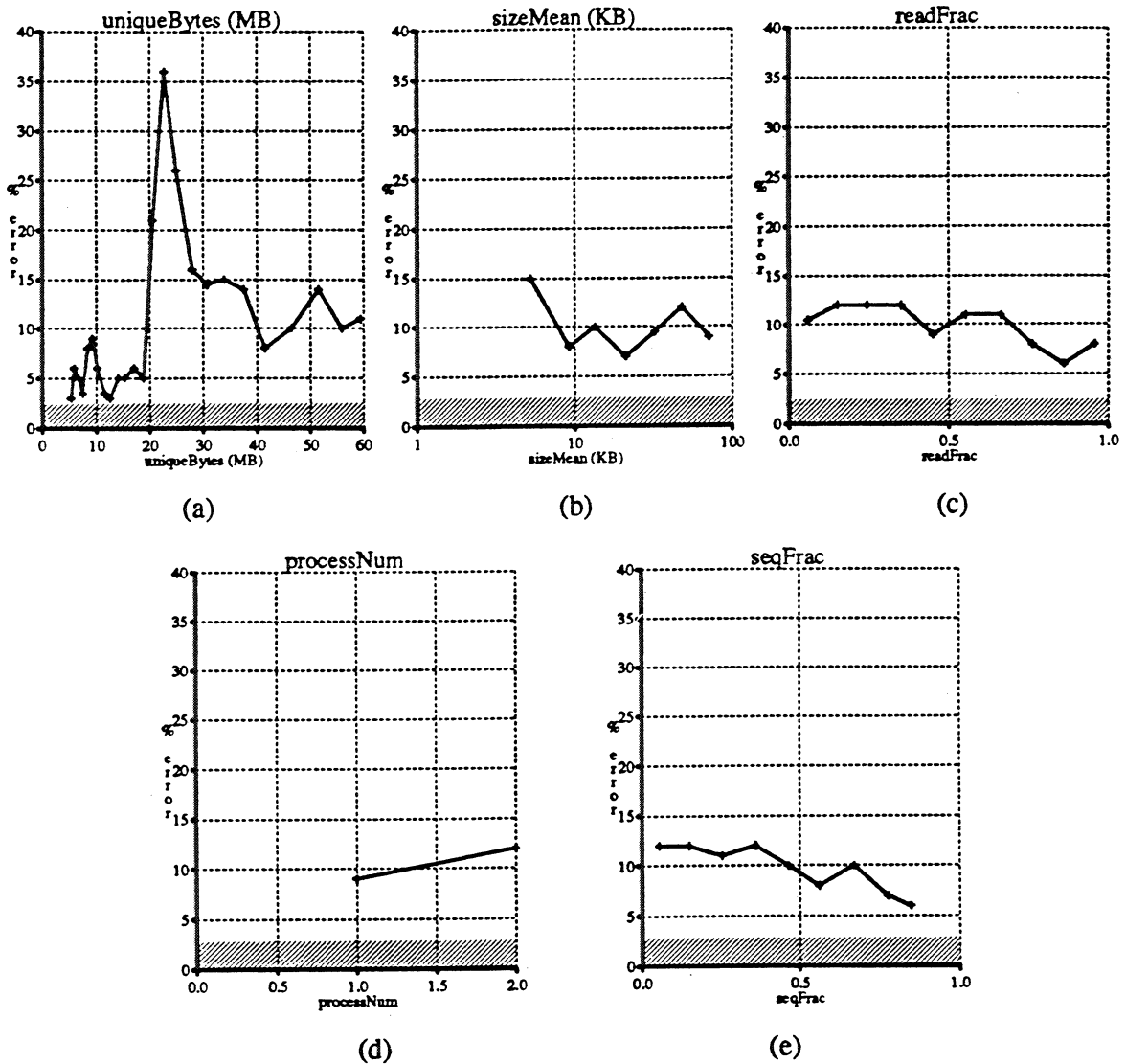
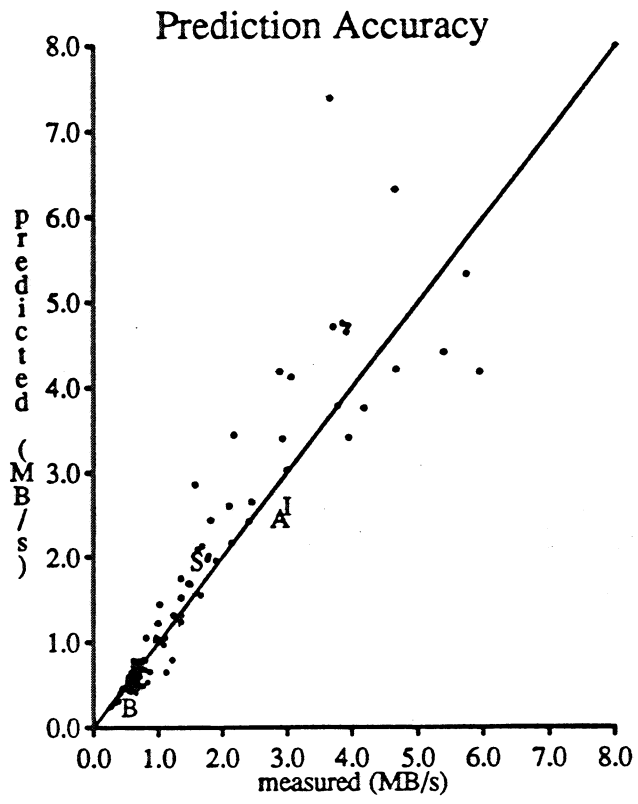
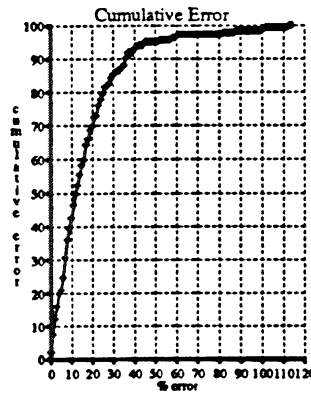


Figure 5.4: What parameters cause errors for SPARCstation 1+. This figure shows the correlation between parameter values and prediction error for the SPARCstation 1+. The vertical line in graph (a) indicates the effective size of the file cache. The shaded region indicates the inherent measurement error due to the variability in I/O performance observed between experiments. Error is most closely correlated to the value of uniqueBytes, and prediction is particularly poor near the border between performance regions. As expected, sharp drops in performance lead to unstable throughput and poor prediction.



(a)



(b)

**Figure 5.5: Evaluation of prediction accuracy for Sprite DECstation 5000/200.** This figure graphs the predicted performance against the actual (measured) performance for the DECstation 5000/200 in Figure 4.8. Points representing traditional benchmarks are shown as the letters "A", "B", "I", and "S", representing Andrew, Bonnie, IOStone, and Sdet, respectively. Median error is 12%. Performance for each workload ranges from 0.3 MB/s to 6 MB/s.

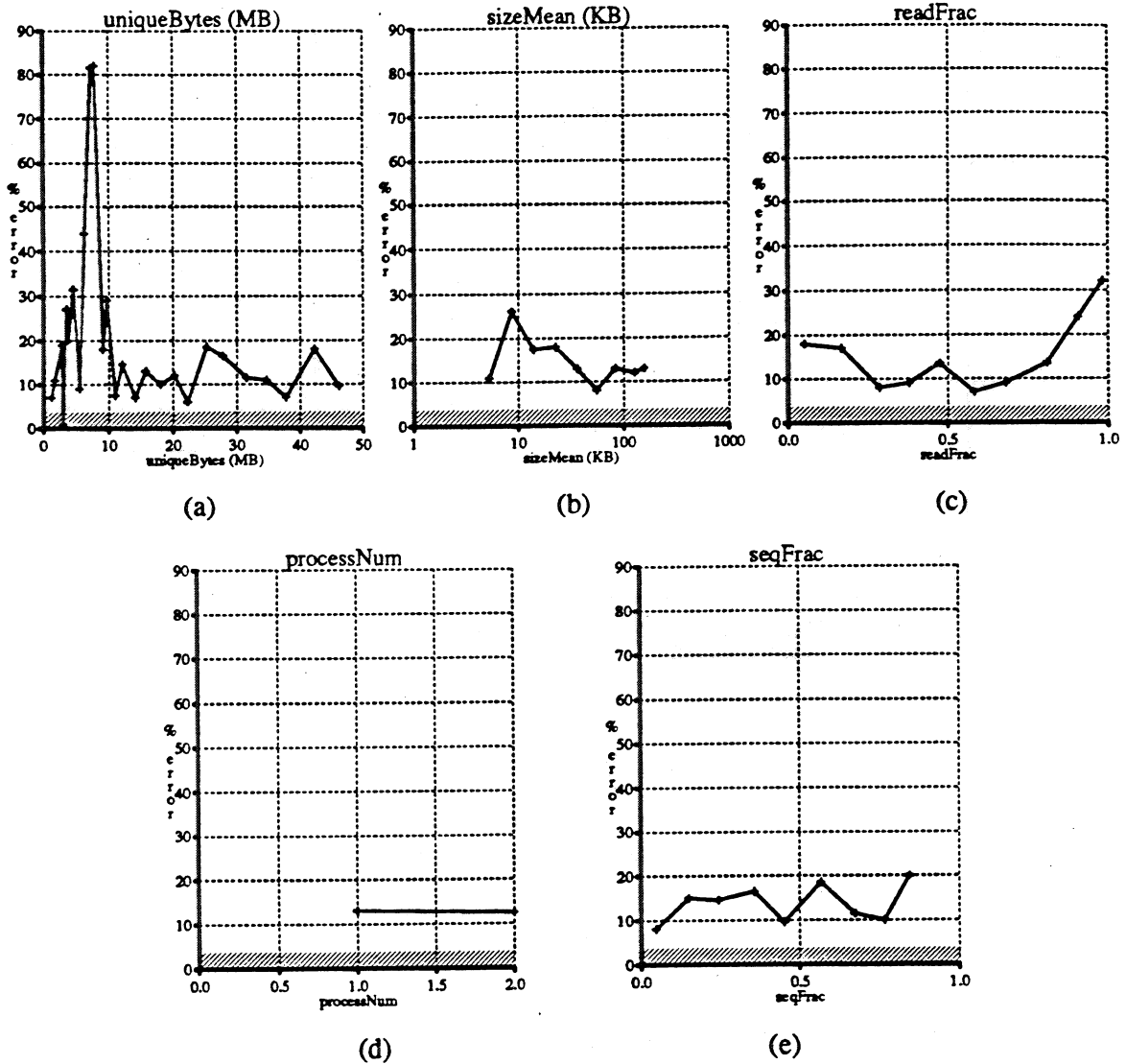
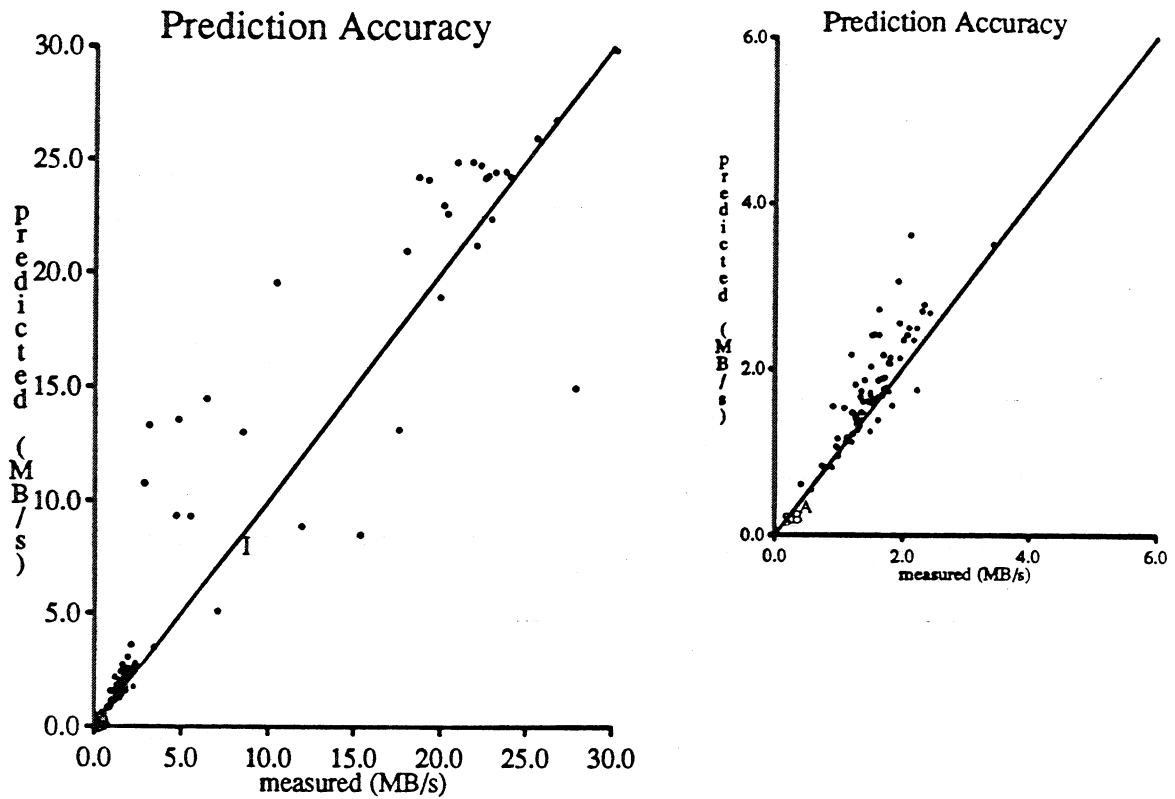
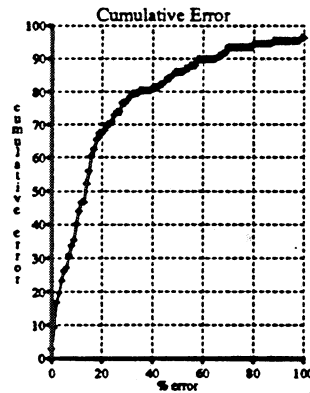


Figure 5.6: What parameters cause errors for DECstation 5000/200. This figure shows the correlation between parameter values and prediction error for the Sprite DECstation 5000/200. The vertical lines in graph (a) indicate the effective size of the file cache, one line for reads, the other for writes. The shaded region indicates the inherent measurement error due to the variability in I/O performance observed between experiments. Again, error is most closely correlated to the value of uniqueBytes; prediction is particularly poor near the border between performance regions. As expected, sharp drops in performance lead to unstable throughput and poor prediction.



(a)



(b)

Figure 5.7: Evaluation of prediction accuracy for HP 730. This figure graphs the predicted performance against the actual (measured) performance for the HP 730 in Figure 4.10 (a closeup of the 0 to 6 MB/s range is shown to the right). Points representing traditional benchmarks are shown as the letters "A", "B", "I", and "S", representing Andrew, Bonnie, IOStone, and Sdet, respectively. Median error is 13%. Performance for each workload ranges from 0.5 MB/s to 31 MB/s.



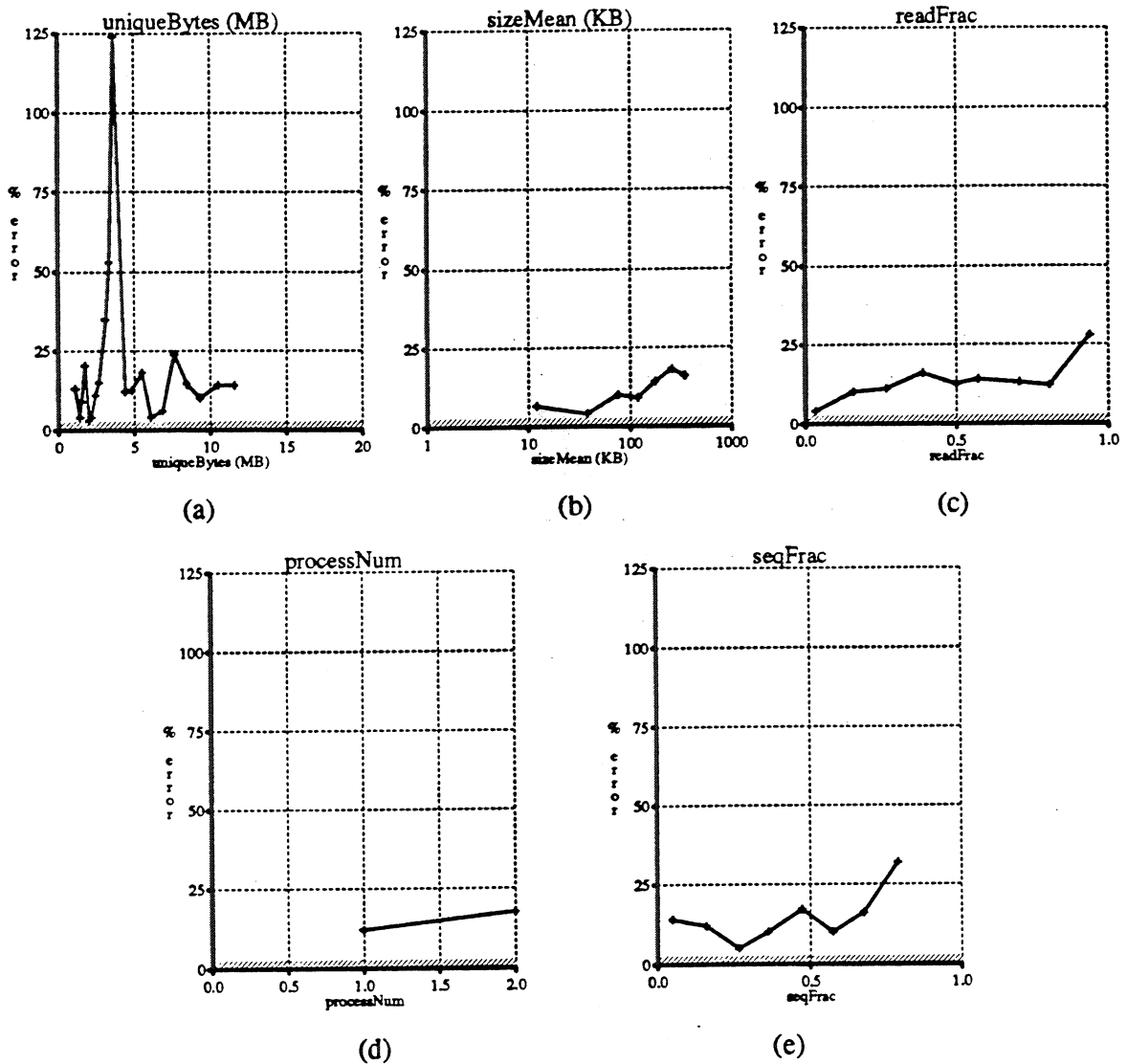
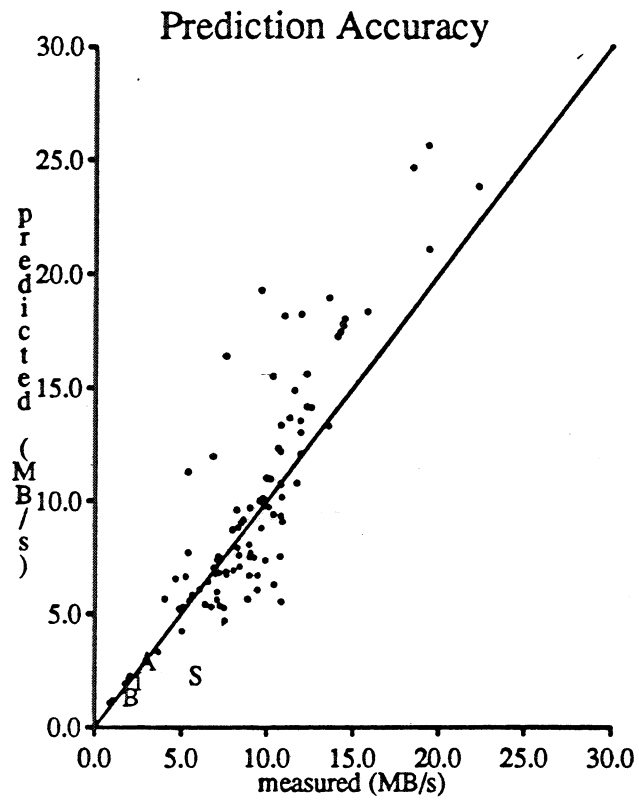
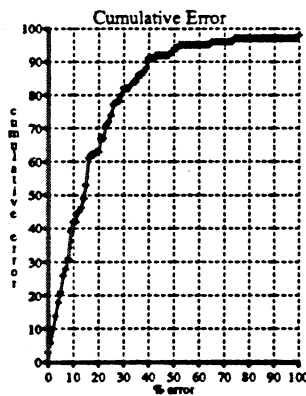


Figure 5.8: What parameters cause errors for HP 730. This figure shows the correlation between parameter values and prediction error for the HP 730. The vertical line in graph (a) indicates the effective size of the file cache. The shaded region indicates the inherent measurement error due to the variability in I/O performance observed between experiments. Again, error is most closely correlated to the value of uniqueBytes; prediction is particularly poor near the border between performance regions. As expected, sharp drops in performance lead to unstable throughput and poor prediction.



(a)



(b)

**Figure 5.9: Evaluation of prediction accuracy for Convex C240.** This figure graphs the predicted performance against the actual (measured) performance for the Convex C240 in Figure 4.11. Points representing traditional benchmarks are shown as the letters "A", "B", "I", and "S", representing Andrew, Bonnie, IOStone, and Sdet, respectively. Median error is 15%.

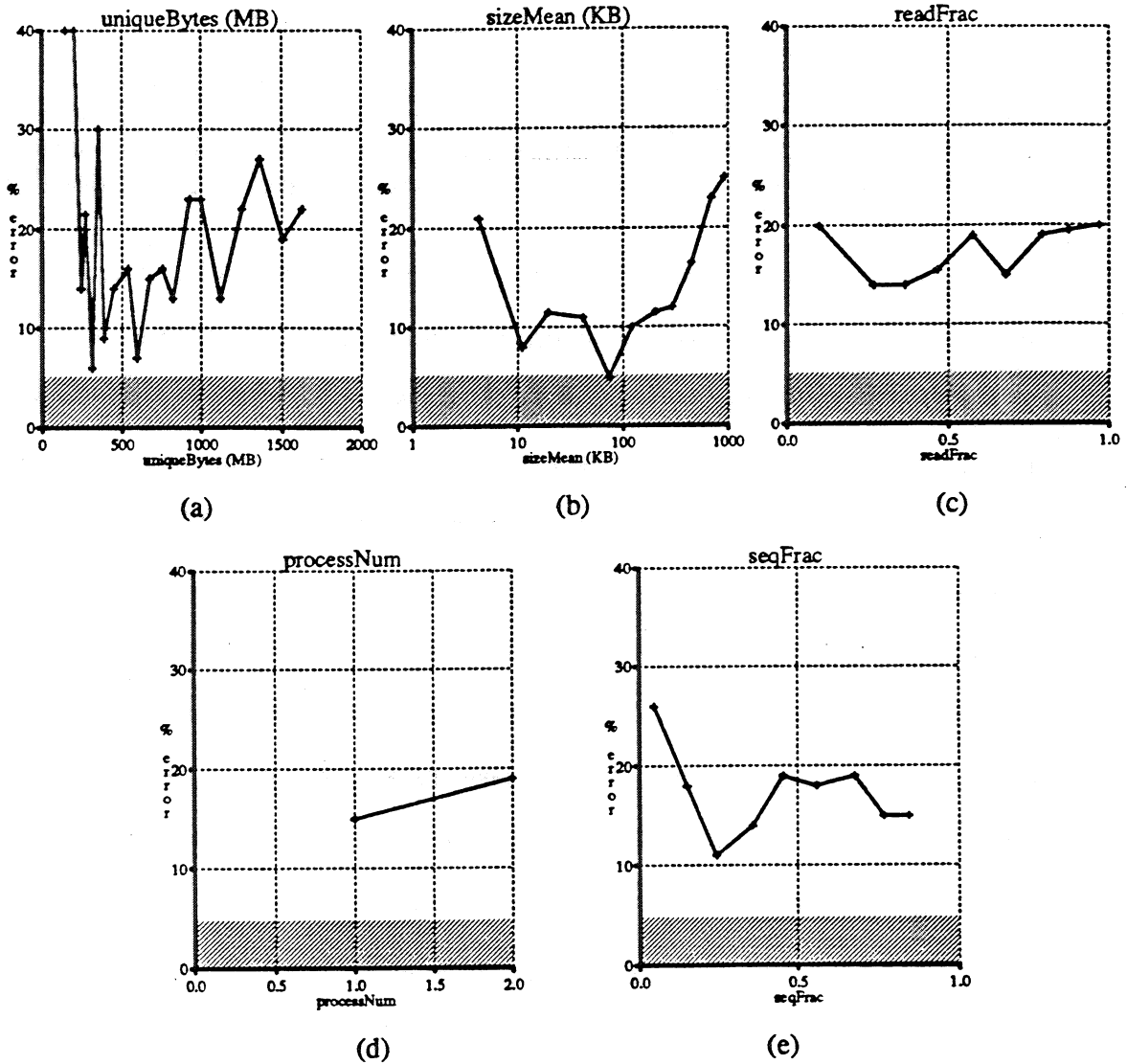


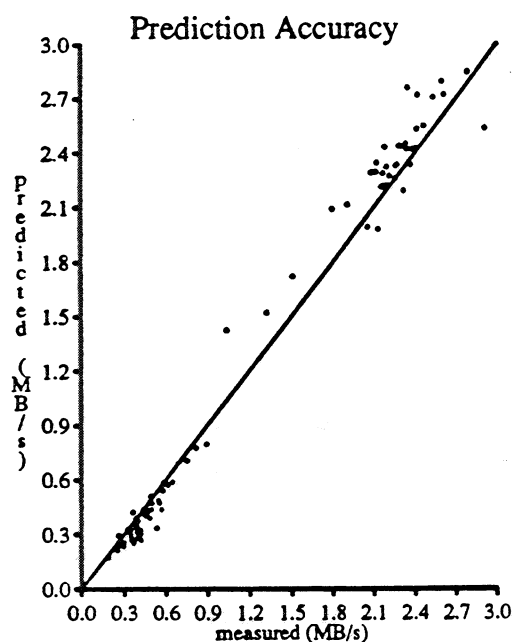
Figure 5.10: What parameters cause errors for Convex C240. This figure shows the correlation between parameter values and prediction error for the Convex C240. The vertical line in graph (a) indicates the effective size of the file cache. The shaded region indicates the inherent measurement error due to the variability in I/O performance observed between experiments. Error is most closely related to size-Mean. Because the Convex supports much larger sizes without degrading performance, it is a more challenging machine for which to predict performance. However, prediction when size is less than 300 KB is quite good (median error of 8%).

System	Median Error	Repeatability Error
SPARCstation 1+	10%	2%
DECstation 5000/200	12%	3%
HP 730	13%	3%
Convex C240	15%	5%

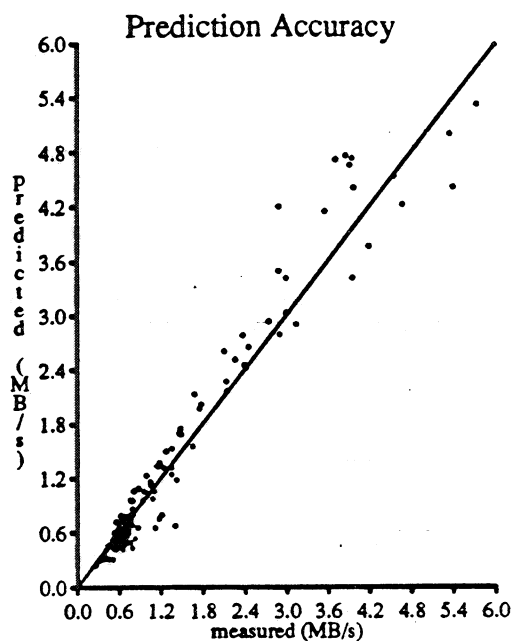
**Table 5.1: Summary of median prediction errors.** This table summarizes the prediction errors on all systems. The third column in the table lists the inherent measurement error, which was measured by running the same set of random workloads twice and using one run to “predict” performance of the other run.

prediction is poor when uniqueBytes is on the border between performance regions (6-10 MB). For the HP 730, this region of high error occurs between 3 and 4 MB. The HP 730’s exceptionally large drop in performance between the file cache and disk region highlights this high error region. Note in Figure 5.7 how most points of high error have performances between 5-20 MB/s. This level of performance can only occur when the system’s file cache is starting to thrash, which is when uniqueBytes is between 3 and 4 MB. Figure 5.11c shows the enhanced prediction when workloads with uniqueBytes in this range are not plotted. Median error drops from 13% to 8%, and all the points of high error in the 5-20 MB/s region of the scatter plot are no longer present. Figure 5.11a 5.11b show revised scatter plots for the SPARCstation 1+ and the DECstation 5000/200 without their workloads with uniqueBytes in the ranges 20-27 MB and 5-10 MB, respectively.

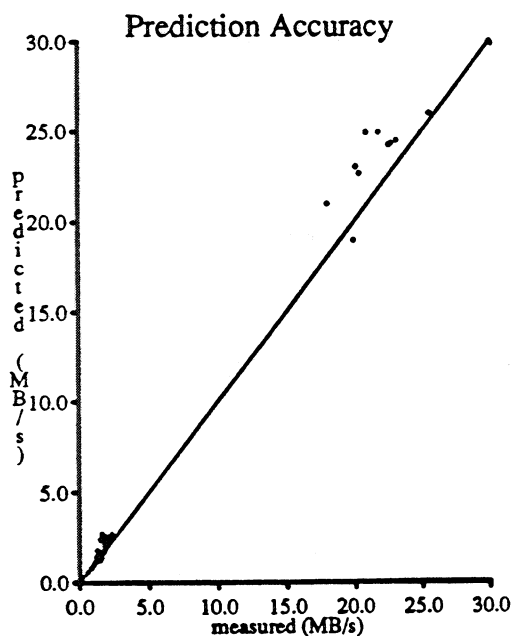
For the Convex C240, the amount of error shows more correlation to sizeMean than to uniqueBytes (Figure 5.10). Because the Convex supports larger sizes before degrading in performance, the self-scaling benchmark chooses a wider range over which to graph sizeMean. When sizes are restricted to a range similar to that of the other systems, prediction accuracy improves to a corresponding level. Figure 5.11d shows a scatter plot without workloads with sizes larger than 300 KB; median error here is only 8%. This indicates that the range of accurate prediction from each focal point may be limited, though not severely. I anticipate that as systems continue to evolve, the range of I/O workloads they support well will widen, forcing any performance prediction to gather and use more points to accurately characterize this



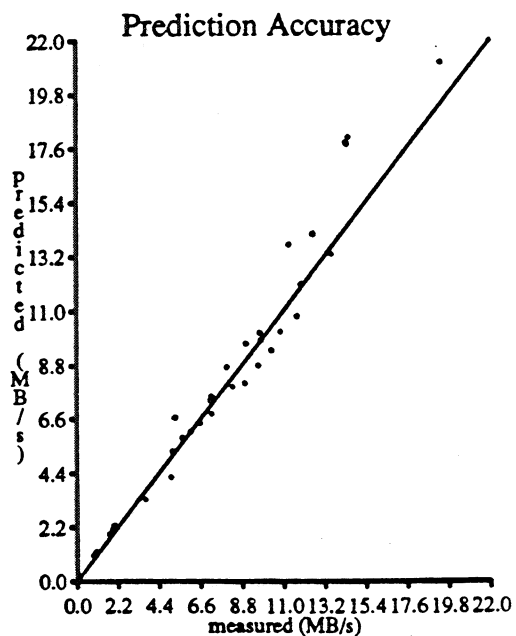
(a) SPARCstation 1+



(b) DECstation 5000/200



(c) HP 730



(d) Convex C240

**Figure 5.11: Enhanced prediction accuracy.** This figure shows the enhanced prediction accuracy when the points of high error are taken out of Figures 5.3, 5.5, 5.7, and 5.9. For the SPARCstation 1+, workloads with uniqueBytes between 20 and 27 MB are taken out and median error drops from 10% to 7%. For the DECstation 5000/200, workloads with uniqueBytes between 5 and 10 MB are taken out and median error drops from 12% to 10%. For the HP 730, workloads with uniqueBytes between 3 and 4 MB are taken out and median error drops from 13% to 8%. For the Convex C240, workloads with very large mean sizes (greater than 300 KB) are taken out and median error drops from 14% to 8%.

increased range. Perhaps the self-scaling benchmark will need to measure and graph more focal points to cover this wider range; perhaps the self-scaling benchmark will need to get feedback from performance prediction as it chooses focal points.

#### 5.4. Comparison Against Orthogonal Sampling

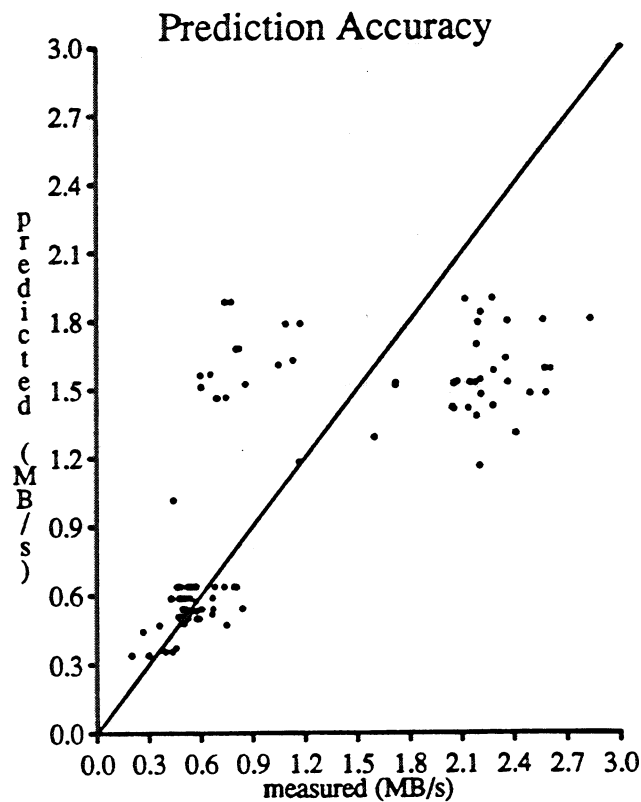
The section above showed how a simple *product of single parameter functions* model can predict performance much more accurately than can single-point benchmarks. However, the *product of single parameter functions* method of prediction uses many more workloads than do single-workload benchmarks such as Andrew. Perhaps the increased accuracy shown in the previous section is due entirely to using more points and not to the validity of the independent functions model. In this section, I explore the accuracy of predicting performance using more workload points but *not* using the *product of single parameter functions* approach. This prediction is done by using multi-dimensional interpolation on an orthogonal sampling of workloads. Orthogonal sampling means selecting a fixed number of evenly spaced values from each parameter's range and measuring all combinations of workloads with those parameter values. For example, assume that a workload consists of two parameters, `readFrac`, ranging from 0-1, and `uniqueBytes`, ranging from 1 MB to 9 MB. Taking three values from each parameter would lead to an orthogonal sampling of nine total workloads of (`readFrac`, `uniqueBytes`) pairs: (0, 1MB), (0, 5 MB), (0, 9 MB), (0.5, 1MB), (0.5, 5 MB), (0.5, 9 MB), (1, 1MB), (1, 5 MB), and (1, 9 MB).

For the SPARCstation 1+, my original approach of using a *product of single parameter functions* requires the 84 points, each taking approximately 10 minutes to measure, that are returned by the self-scaling benchmark<sup>2</sup>. As can readily be seen, orthogonal sampling

---

<sup>2</sup>Six of the nine graphs have 10 points each; one (`uniqueBytes`) uses 20 points; two (`processNum`) use only two points each, due to the limited range of `processNum`.

inherently requires many more points. Sampling three values per dimension requires 162 points<sup>3</sup>. Figure 5.12 shows that even with twice as many points, interpolation using an orthogonal sampling gives much worse prediction accuracy than does the *product of single parameter functions*: the median error using orthogonal sampling is 22% versus 10% using the *product of single parameter functions*, and it takes twice as long to collect the necessary information.



**Figure 5.12: Prediction accuracy using interpolation on an orthogonal sample.** This figure shows that interpolation using an orthogonal sample gives poorer prediction than does a prediction model based on *products of single parameter functions*. Even when using twice as many workloads to do prediction, median error is twice as high (22%).

---

<sup>3</sup>Normally this would be  $3^5$ , or 243 points. However, due to the limited range of processNum, only two points are required for this dimension, which brings the total required to 162.

## 5.5. Application of Predicted Performance—Performance Ratios

Performance evaluators sometimes want to measure more than absolute performance. Their end goal is often to compare two systems head-to-head, generally as the ratio of performances. In this section, I apply the model of performance prediction developed in Section 5.2 to predict the performance ratio between several systems.

To measure and predict performance ratios, I define several workloads to be run on all systems. Table 5.2 describes these workloads and gives example applications for each. Figure 5.13 shows the measured and predicted ratios for these workloads. For the sake of comparison, these figures also show the performance ratio given by Andrew.

Though single-point benchmarks do not predict absolute performance well for a wide range of workloads, they could in theory still accurately estimate the ratio of two machines' performance. For this to be true, the ratio of two systems' performance would need to be constant and independent of the workload. This is not the case, however. Figure 5.13 shows that the performance ratio of two systems can vary greatly. For example, Figure 5.13a shows that for the large utility workload, a DECstation 5000 is five times faster than an HP 730; however, for the workstation and scientific read workloads, an HP 730 is three times faster than a DECstation 5000. Predicting performance using the output of the self-scaling benchmark captures this variability in the ratio of two systems' performance in a way that no single-point benchmark can. In addition, once the results from the self-scaling benchmark are in hand, prediction for any number of workload takes essentially no time. In contrast, measuring just the small set of workloads in Figure 5.13 took 2-3 hours for each system.

## 5.6. Conclusions

I have shown in this chapter that a simple *product of single parameter functions* approach to performance prediction is highly accurate, an outcome that has two important ramifications.



Title	Example	unique Bytes (MB)	seq Frac	read Frac	sizeMean (KB)	process Num
workstation	compile grep	1	0.8	0.8	4	1
large_utility	sort	10	0.9	0.6	8	1
scientific_write	satellite data	250	0.5	0.2	100	1
scientific_read	image process	250	0.5	0.8	100	1
database	TPC	500	0.1	0.2	4	2

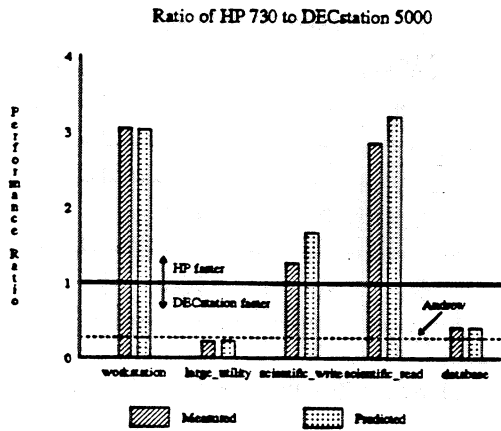
**Table 5.2: Workloads to be run on all systems.** This table describes the workloads I use when comparing two systems.

First, it makes the graphs from the self-scaling benchmark much more useful, since they can be used to estimate performance for other workloads. Second, it supports the assumption of shape independence, that is, the shape of each graph from the self-scaling benchmarks is approximately independent of the values of the other parameters.

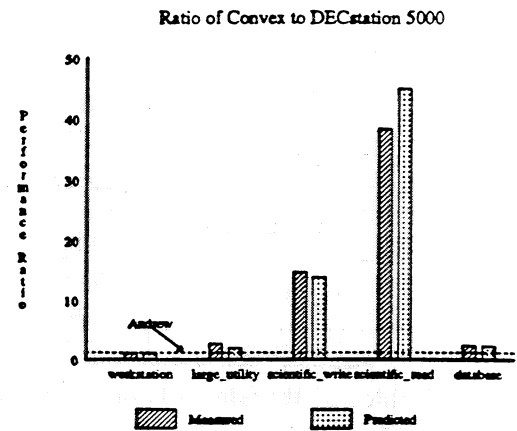
Section 5.5 applies predicted performance to the traditional use of benchmarking, which compares the performance of two systems. Performance prediction can be used to accurately predict the ratio of performance between two systems on identical workloads, even when that workload has not been run on either system. Single-workload benchmarks, in contrast, give poor estimates of performance ratios except for their own workloads.

The enhanced performance prediction over single-point benchmarks comes partly from using more measurements; however, the *product of single parameter functions* approach is equally, if not more, important. This was shown when interpolation could not predict performance accurately, even when using twice as many measured workloads.

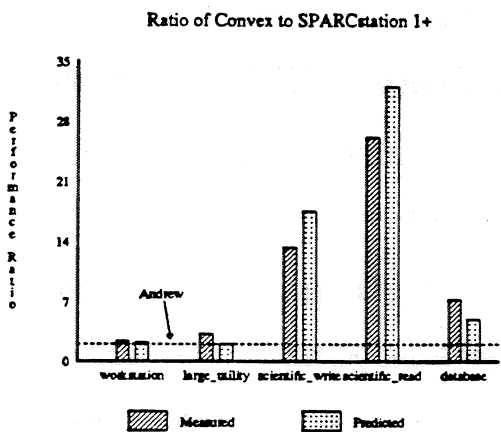
The payoff for running the time consuming, self-scaling benchmark comes from predicted performance. The self-scaling benchmark need only be run once per configuration. This means either a manufacturer or an industry-wide performance group, such as SPEC, need only run the data once and then publish it for everyone else to use. Each possible workload can then be



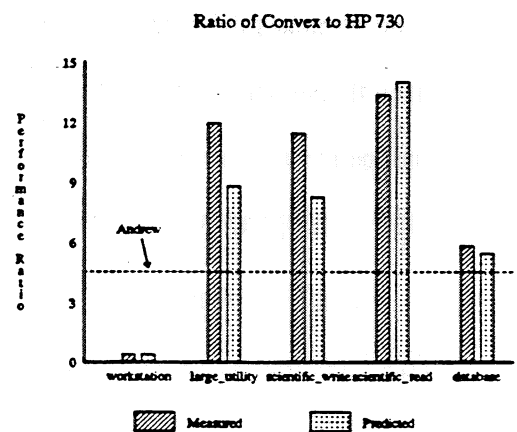
(a)



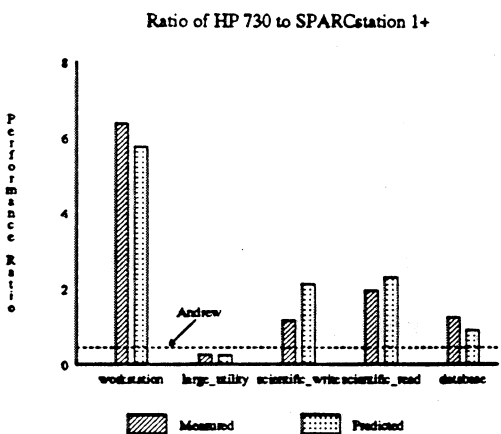
(b)



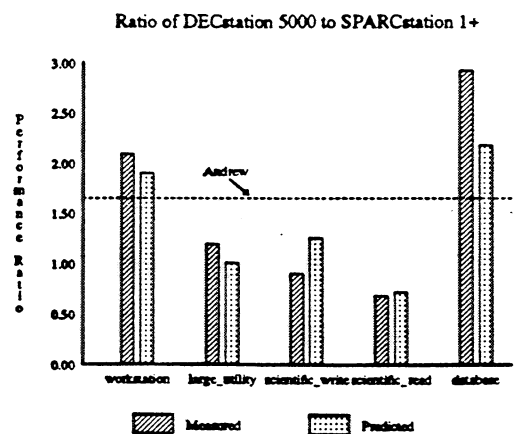
(c)



(d)



(e)



(f)

Figure 5.13: Measured versus predicted ratio. This figures shows how accurately systems can be compared using performance prediction. For comparison, the performance ratio given by Andrew is shown as a dashed line. Predicting performance using the output of the self-scaling benchmark captures this variability in the ratio of two systems' performance in a way that no single-point benchmark can.

predicted in milliseconds.

## 5.7. References

[Saavedra-Barrera89]

R. H. Saavedra-Barrera, A. J. Smith and E. Miya, "Machine Characterization Based on an Abstract High-Level Language Machine", *IEEE Transactions on Computers* 38, 12 (December 1989), 1659-1679.

---

## Chapter 6

# Conclusions and Future Work

---

### 6.1. Conclusions

In this thesis, I proposed a new approach to I/O performance evaluation—self-scaling benchmarks and predicted performance. This thesis' self-scaling benchmark yields information not only about performance but also about what workloads are appropriate for the system. The self-scaling benchmark scales automatically to current and future machines by adapting the workload to the system being tested. Further, it gives insight into the system's performance characteristics by revealing the performance dependencies for each of five workload parameters. I used the self-scaling benchmark on nine systems, ranging from a single-disk workstation to a striped-disk mini-supercomputer, and gained insight about appropriate block sizes, file cache sizes, read/write behavior of the caches, and the benefits of sequentiality.

My first attempt to make a self-scaling benchmark showed that scaling all workload parameters was too extreme. However, scaling some parameters created a highly useful evaluation tool, as the modified self-scaling benchmark showed.

In the process of using the self-scaling benchmark, I found that many systems are not tailored to handle I/O-intensive workloads. Two systems that were designed specifically to be file servers (a Solbourne and an Auspex file server) crashed or deadlocked while running the self-scaling benchmark. Clearly, much more needs to be done to make systems more robust since more I/O-intensive workloads are forthcoming.

Predicted performance restores the ability to compare two machines on the same workload lost in the self-scaling benchmark. Further, it extends this ability to workloads that have not been measured by estimating performance based on the graphs from the self-scaling benchmark. I have shown that this prediction is far more accurate over a wide range of workloads than any single-point benchmark, both in absolute performance and relative performance of two systems. In addition, prediction using the *product of single parameter functions* model gives much more accurate performance estimates than does interpolation using even twice as many measured workload points.

The accuracy of my method of predicting performance shows that it is possible to characterize the performance of a wide range of workloads accurately with a few tens of workload points. I hope that the added utility of being able to apply the performance graphs of the self-scaling benchmark to such a wide range of workloads will overcome the industry's tendency to focus on the performance of a single workload.

Self-scaling benchmarks and predicted performance could fundamentally affect how manufacturers and users view I/O evaluation in four critical ways. First, it condenses performance over a wide range of workloads into a few graphs. If manufacturers or standard performance groups (such as SPEC or computer magazines) were to publish such graphs, users could

use predicted performance to estimate, *without further measurements*, the I/O performance of their specific workloads. Ideally, manufacturers would go further and publish results from the self-scaling benchmark for many different system configurations, such as for different numbers of disks and main memory sizes, along with the cost of each configuration. With this information, users could calculate the price/performance ratios for each configuration *on their specific workloads* and make much more informed purchasing decisions.

Second, by taking advantage of self-scaling benchmark's ease of use, manufacturers could easily evaluate many I/O configurations. Instead of merely reporting performance for each I/O configuration on a few workloads, the self-scaling benchmark would report both the performance for many workloads and the I/O workloads that perform well under this configuration. Hence, manufacturers could better identify each product's target application area. Because the price of each configuration can easily be calculated, the price/performance of systems that match the users needs can also easily be calculated. This can help buyers make choices such as whether to purchase many small disks or a few large large disks, more memory and a larger file cache or faster disks, and so on.

Third, system developers could benefit by using the self-scaling benchmark to understand the effects of any hardware and software changes. Unlike traditional, single-point benchmarks, these effects would be shown over a wide range of workloads.

Lastly, the self-scaling benchmark introduces a new approach to scaling—scaling aspects of the workload to the changing performance characteristics of systems. This approach has the potential to successfully scale with tomorrow's faster and larger I/O systems.

## **6.2. Future Work**

More work needs to be done in the general area of self-scaling benchmarks. In the future, I/O systems will support a wider range of workloads and stretch the limits of predicting performance. Perhaps more focal points will be needed to characterize performance over this

increased range of workloads. It would also be interesting to use the self-scaling benchmark on a wider variety of systems. I see four dimensions that would be fruitful to explore with self-scaling benchmarks. First, more classes of systems need to be measured. PCs, mainframes, and multiprocessors could each add insights to the usefulness of self-scaling benchmarks as well as validity to the prediction model described in Chapter 5. Second, more network-based client-server systems should be measured, because these types of file servers are becoming more prevalent. Third, the self-scaling benchmark could be used on a wider range of I/O systems, including solid-state and optical disks, magnetic tapes, and systems with file migration. Fourth, non-Unix operating systems would be interesting to measure. Each of these dimensions of different types of systems would demonstrate the utility of self-scaling benchmarks or would serve to further refine the self-scaling ideas. Self-scaling benchmarks could also be applied to non-I/O areas such as processor performance. I suspect this will prove much harder since synthetic workloads for processors are notoriously bad at accurately representing real applications.

In the area of predicted performance, researching ways to enhance prediction accuracy or to better characterize the workload areas that yield high error would be beneficial. One can also imagine a completely different approach to performance prediction. Instead of the current scheme, which separates the self-scaling benchmark's measurement phase from the performance prediction phase, I could introduce feedback between the two phases. For example, the self-scaling benchmark could choose random workloads in the workload space and make sure the current set of focal points accurately predicts performance. When a workload is found for which the current set of focal points does not accurately predict performance, a new focal point could be added to predict that workload. Although this would lead to longer running times for the benchmark, it may enhance prediction.

This thesis has brought up the need for more realistic, flexible benchmarks, that is, benchmarks that can be modified to create many different workloads (such as Willy) but still retain a

link to real-world applications. One possible candidate is a *flexible trace*, where a trace of a real application is modified to have different characteristics (such as changing the number of processes or the average request size).

In conclusion, computer performance evaluation must continue to develop novel ways to stress our rapidly evolving new systems. Self-scaling benchmarks and predicted performance can play a key role in this process because they have the potential to break the weary habit of using obsolete benchmarks by automatically keeping benchmarks up-to-date with new generations of systems.



---

# Appendix A

## Data Used in Prediction

---

This appendix gives the raw data used in the scatter plots of Chapter 5. To allow the table to fit on the page, I use the following abbreviations:

un = uniqueBytes in MB

seq = seqFrac

read = readFrac

sz = sizeMean in KB

proc = processNum

Measured = Measured Performance in MB/s

Predicted = Predicted Performance in MB/s

% Err = % Error

Data for SPARCstation 1+ (Figure 5.3) — Part 1

Workload	un	seq	read	sz	proc	Measured	Predicted	% Err
Andrew	4.7	.77	.54	3	1	1.5 MB/s	1.5 MB/s	5 %
Bonnie	100	1.0	0.0	8	1	.38 MB/s	.27 MB/s	29 %
IOStone	1.0	.02	.67	2	1	1.6 MB/s	1.6 MB/s	0 %
Sdet	8.0	.48	.56	2	4	1.2 MB/s	1.3 MB/s	11 %
Random 0	53	0.73	0.21	25	1	0.4 MB/s	0.4 MB/s	4 %
Random 1	33	0.39	0.71	26	1	0.6 MB/s	0.5 MB/s	14 %
Random 2	9	0.58	0.46	51	2	2.3 MB/s	2.5 MB/s	5 %
Random 3	46	0.37	0.25	82	2	0.4 MB/s	0.3 MB/s	28 %
Random 4	10	0.28	0.01	46	1	2.4 MB/s	2.5 MB/s	5 %
Random 5	40	0.80	0.82	29	2	0.5 MB/s	0.5 MB/s	4 %
Random 6	14	0.22	0.16	67	1	2.4 MB/s	2.4 MB/s	1 %
Random 7	21	0.82	0.44	68	2	1.8 MB/s	1.4 MB/s	21 %
Random 8	54	0.72	0.06	83	1	0.5 MB/s	0.4 MB/s	10 %
Random 9	41	0.13	0.12	43	1	0.4 MB/s	0.3 MB/s	13 %
Random 10	19	0.29	0.04	74	2	2.1 MB/s	2.0 MB/s	7 %
Random 11	24	0.04	0.26	76	1	0.7 MB/s	0.7 MB/s	5 %
Random 12	34	0.04	0.53	39	1	0.5 MB/s	0.4 MB/s	10 %
Random 13	10	0.85	0.67	65	2	2.2 MB/s	2.4 MB/s	11 %
Random 14	5	0.51	0.76	37	2	2.6 MB/s	2.7 MB/s	4 %
Random 15	53	0.65	0.16	75	1	0.5 MB/s	0.4 MB/s	9 %
Random 16	35	0.19	0.99	35	1	0.6 MB/s	0.6 MB/s	9 %
Random 17	23	0.67	0.76	70	2	1.2 MB/s	1.1 MB/s	11 %
Random 18	41	0.73	0.01	79	1	0.5 MB/s	0.5 MB/s	6 %
Random 19	57	0.16	0.94	6	1	0.3 MB/s	0.3 MB/s	11 %
Random 20	35	0.81	0.76	50	1	0.8 MB/s	0.7 MB/s	6 %
Random 21	12	0.23	0.28	22	1	2.3 MB/s	2.3 MB/s	3 %
Random 22	25	0.27	0.74	56	1	1.0 MB/s	0.8 MB/s	23 %
Random 23	36	0.40	0.59	26	2	0.4 MB/s	0.3 MB/s	17 %
Random 24	26	0.55	0.16	70	2	0.7 MB/s	0.5 MB/s	18 %
Random 25	55	0.24	0.57	18	2	0.3 MB/s	0.2 MB/s	10 %
Random 26	7	0.16	0.69	33	1	2.4 MB/s	2.8 MB/s	18 %
Random 27	26	0.48	0.10	47	1	0.6 MB/s	0.6 MB/s	7 %
Random 28	11	0.79	0.04	16	1	2.1 MB/s	2.3 MB/s	8 %
Random 29	46	0.79	0.20	18	1	0.4 MB/s	0.4 MB/s	7 %
Random 30	21	0.71	0.59	62	2	1.7 MB/s	1.2 MB/s	29 %
Random 31	56	0.15	0.53	7	2	0.2 MB/s	0.2 MB/s	7 %
Random 32	22	0.84	0.85	36	1	2.0 MB/s	1.7 MB/s	16 %
Random 33	34	0.42	0.53	61	2	0.5 MB/s	0.4 MB/s	19 %
Random 34	17	0.34	0.45	34	2	2.2 MB/s	2.2 MB/s	2 %
Random 35	11	0.49	0.51	81	2	2.2 MB/s	2.2 MB/s	1 %
Random 36	52	0.22	0.37	22	2	0.3 MB/s	0.2 MB/s	20 %
Random 37	39	0.75	0.55	21	2	0.4 MB/s	0.4 MB/s	5 %
Random 38	42	0.42	0.43	46	2	0.4 MB/s	0.3 MB/s	13 %
Random 39	48	0.40	0.86	81	2	0.4 MB/s	0.4 MB/s	0 %
Random 40	39	0.17	0.85	33	1	0.5 MB/s	0.5 MB/s	11 %
Random 41	28	0.65	0.42	32	1	0.6 MB/s	0.6 MB/s	0 %
Random 42	48	0.12	0.16	29	1	0.3 MB/s	0.3 MB/s	15 %
Random 43	53	0.06	0.12	49	2	0.4 MB/s	0.3 MB/s	31 %
Random 44	9	0.60	0.26	26	2	2.3 MB/s	2.4 MB/s	5 %

Data for SPARCstation 1+ (Figure 5.3) — Part 2

Workload	un	seq	read	sz	proc	Measured	Predicted	% Err
Random 45	11	0.65	0.56	57	1	2.4 MB/s	2.4 MB/s	2 %
Random 46	38	0.71	0.88	20	2	0.5 MB/s	0.5 MB/s	1 %
Random 47	39	0.81	0.87	57	1	0.8 MB/s	0.8 MB/s	5 %
Random 48	19	0.42	0.46	82	2	2.1 MB/s	2.0 MB/s	3 %
Random 49	38	0.33	0.32	71	2	0.4 MB/s	0.3 MB/s	23 %
Random 50	41	0.53	0.52	19	2	0.3 MB/s	0.3 MB/s	0 %
Random 51	53	0.26	0.30	15	1	0.3 MB/s	0.3 MB/s	15 %
Random 52	18	0.06	0.65	28	2	2.2 MB/s	2.2 MB/s	2 %
Random 53	45	0.09	0.19	58	2	0.4 MB/s	0.3 MB/s	29 %
Random 54	13	0.41	0.20	55	2	2.3 MB/s	2.3 MB/s	3 %
Random 55	16	0.53	0.72	30	2	2.2 MB/s	2.3 MB/s	3 %
Random 56	16	0.47	0.47	4	2	1.0 MB/s	1.4 MB/s	37 %
Random 57	48	0.08	0.54	79	2	0.4 MB/s	0.3 MB/s	20 %
Random 58	13	0.82	0.77	78	2	2.1 MB/s	2.3 MB/s	10 %
Random 59	15	0.14	0.31	4	1	1.3 MB/s	1.5 MB/s	15 %
Random 60	48	0.69	0.22	49	1	0.5 MB/s	0.4 MB/s	11 %
Random 61	53	0.10	0.92	43	2	0.4 MB/s	0.4 MB/s	1 %
Random 62	12	0.12	0.63	61	1	2.3 MB/s	2.4 MB/s	4 %
Random 63	9	0.72	0.33	5	1	1.5 MB/s	1.7 MB/s	14 %
Random 64	17	0.19	0.11	79	2	2.1 MB/s	2.3 MB/s	10 %
Random 65	22	0.23	0.86	80	1	2.5 MB/s	1.4 MB/s	43 %
Random 66	8	0.67	0.89	63	2	2.8 MB/s	2.8 MB/s	2 %
Random 67	21	0.08	0.41	30	1	2.2 MB/s	1.5 MB/s	32 %
Random 68	42	0.69	0.57	65	2	0.4 MB/s	0.4 MB/s	4 %
Random 69	30	0.38	0.98	61	1	0.9 MB/s	0.8 MB/s	11 %
Random 70	39	0.47	0.13	27	1	0.4 MB/s	0.4 MB/s	3 %
Random 71	52	0.05	0.12	73	2	0.4 MB/s	0.3 MB/s	36 %
Random 72	10	0.76	0.24	19	2	2.2 MB/s	2.3 MB/s	6 %
Random 73	7	0.26	0.67	52	2	2.5 MB/s	2.7 MB/s	7 %
Random 74	18	0.77	0.14	26	2	1.8 MB/s	2.1 MB/s	16 %
Random 75	28	0.27	0.69	25	2	0.6 MB/s	0.4 MB/s	23 %
Random 76	30	0.19	0.19	35	1	0.5 MB/s	0.4 MB/s	12 %
Random 77	31	0.25	0.02	78	2	0.5 MB/s	0.3 MB/s	37 %
Random 78	49	0.25	0.39	13	2	0.3 MB/s	0.2 MB/s	15 %
Random 79	9	0.54	0.10	70	2	2.3 MB/s	2.4 MB/s	7 %
Random 80	29	0.45	0.68	75	2	0.6 MB/s	0.5 MB/s	6 %
Random 81	22	0.33	0.70	85	1	2.1 MB/s	1.2 MB/s	44 %
Random 82	36	0.42	0.31	63	1	0.5 MB/s	0.4 MB/s	9 %
Random 83	8	0.53	0.97	11	2	2.4 MB/s	2.7 MB/s	12 %
Random 84	6	0.86	0.51	84	2	2.5 MB/s	2.6 MB/s	3 %
Random 85	55	0.54	0.02	34	2	0.4 MB/s	0.3 MB/s	24 %
Random 86	24	0.78	0.85	45	2	1.3 MB/s	0.9 MB/s	34 %
Random 87	13	0.82	0.51	74	1	2.3 MB/s	2.3 MB/s	0 %
Random 88	12	0.57	0.93	40	2	2.6 MB/s	2.8 MB/s	8 %
Random 89	37	0.81	0.19	80	2	0.5 MB/s	0.4 MB/s	12 %
Random 90	58	0.02	0.95	18	1	0.4 MB/s	0.4 MB/s	16 %
Random 91	16	0.02	0.29	63	2	2.2 MB/s	2.3 MB/s	6 %
Random 92	17	0.48	0.31	24	2	1.9 MB/s	2.1 MB/s	11 %
Random 93	40	0.64	0.12	47	1	0.5 MB/s	0.4 MB/s	6 %

Data for SPARCstation 1+ (Figure 5.3) — Part 3

Workload	un	seq	read	sz	proc	Measured	Predicted	% Err
Random 94	15	0.54	0.16	82	1	2.4 MB/s	2.3 MB/s	1 %
Random 95	19	0.11	0.14	30	1	2.3 MB/s	2.2 MB/s	5 %
Random 96	20	0.77	0.05	75	2	1.8 MB/s	1.7 MB/s	9 %
Random 97	19	0.65	0.99	81	1	2.9 MB/s	2.5 MB/s	13 %
Random 98	44	0.06	0.74	25	2	0.3 MB/s	0.3 MB/s	6 %
Random 99	53	0.31	0.16	46	2	0.4 MB/s	0.3 MB/s	29 %

Table A.1: Raw data for SPARCstation 1+ scatter plot. This table gives the data used to plot the current benchmark prediction points in Figures 5.3. Most points of high error have unique-Bytes between 20 and 27 MB.

Data for DECstation 5000/200 (Figure 5.5) — Part 1

Workload	un	seq	read	sz	proc	Measured	Predicted	% Err
Andrew	4.7	.77	.54	3	1	2.9 MB/s	2.5 MB/s	14 %
Bonnie	100	1.0	0.0	8	1	0.5 MB/s	0.3 MB/s	53 %
IOStone	1.0	.02	.67	2	1	3.0 MB/s	2.6 MB/s	12 %
Sdet	8.0	.48	.56	2	4	1.6 MB/s	1.9 MB/s	24 %
Random 0	31	0.80	0.41	55	2	0.7 MB/s	0.7 MB/s	7 %
Random 1	21	0.85	0.57	56	1	1.7 MB/s	1.5 MB/s	6 %
Random 2	32	0.85	0.37	37	2	0.6 MB/s	0.6 MB/s	3 %
Random 3	36	0.08	0.30	157	2	0.6 MB/s	0.6 MB/s	3 %
Random 4	6	0.22	0.97	60	2	5.9 MB/s	4.2 MB/s	30 %
Random 5	45	0.11	0.46	91	1	0.5 MB/s	0.5 MB/s	11 %
Random 6	7	0.22	0.26	103	2	1.6 MB/s	2.9 MB/s	82 %
Random 7	4	0.61	0.01	84	1	2.9 MB/s	4.2 MB/s	45 %
Random 8	25	0.53	0.91	136	2	1.1 MB/s	0.7 MB/s	42 %
Random 9	21	0.38	0.65	40	1	1.8 MB/s	2.0 MB/s	12 %
Random 10	27	0.57	0.77	115	1	0.9 MB/s	0.7 MB/s	26 %
Random 11	19	0.25	0.69	16	2	2.1 MB/s	2.6 MB/s	23 %
Random 12	9	0.61	0.34	143	1	1.4 MB/s	1.7 MB/s	29 %
Random 13	15	0.53	0.17	73	2	0.8 MB/s	1.1 MB/s	29 %
Random 14	28	0.70	0.06	76	2	0.6 MB/s	0.8 MB/s	26 %
Random 15	31	0.03	0.55	43	1	0.5 MB/s	0.5 MB/s	1 %
Random 16	6	0.39	0.15	41	2	2.2 MB/s	3.4 MB/s	58 %
Random 17	14	0.41	0.59	158	1	1.8 MB/s	2.0 MB/s	13 %
Random 18	20	0.34	0.63	73	1	1.7 MB/s	2.1 MB/s	26 %
Random 19	24	0.41	0.29	75	2	0.8 MB/s	0.8 MB/s	2 %
Random 20	42	0.87	0.39	22	2	0.6 MB/s	0.4 MB/s	24 %
Random 21	5	0.59	0.34	66	2	3.1 MB/s	4.1 MB/s	35 %
Random 22	11	0.28	0.44	112	2	1.5 MB/s	1.7 MB/s	15 %
Random 23	14	0.81	0.76	24	2	3.0 MB/s	3.0 MB/s	1 %
Random 24	45	0.31	0.67	35	1	0.4 MB/s	0.4 MB/s	2 %
Random 25	19	0.24	0.93	132	2	4.7 MB/s	4.2 MB/s	10 %
Random 26	40	0.03	0.69	87	2	0.6 MB/s	0.5 MB/s	8 %
Random 27	16	0.43	0.32	16	1	1.3 MB/s	1.2 MB/s	8 %
Random 28	28	0.53	0.15	99	2	0.6 MB/s	0.7 MB/s	16 %

Data for DECstation 5000/200 (Figure 5.5) — Part 2

Workload	un	seq	read	sz	proc	Measured	Predicted	% Err
Random 29	39	0.14	0.82	5	1	0.3 MB/s	0.2 MB/s	10 %
Random 30	9	0.79	0.67	113	1	2.9 MB/s	3.4 MB/s	16 %
Random 31	34	0.06	0.62	66	2	0.6 MB/s	0.5 MB/s	13 %
Random 32	36	0.54	0.66	45	2	0.5 MB/s	0.5 MB/s	8 %
Random 33	3	0.82	0.27	82	1	3.9 MB/s	4.7 MB/s	19 %
Random 34	20	0.59	0.87	83	2	4.0 MB/s	3.4 MB/s	14 %
Random 35	9	0.00	0.38	38	1	1.9 MB/s	2.0 MB/s	3 %
Random 36	10	0.17	0.17	100	2	1.0 MB/s	1.4 MB/s	41 %
Random 37	42	0.78	0.22	154	1	0.6 MB/s	0.6 MB/s	7 %
Random 38	3	0.12	0.34	137	1	3.7 MB/s	4.7 MB/s	27 %
Random 39	24	0.69	0.82	158	2	1.2 MB/s	0.8 MB/s	34 %
Random 40	39	0.43	0.98	139	2	0.7 MB/s	0.5 MB/s	32 %
Random 41	9	0.06	0.53	5	1	1.8 MB/s	2.4 MB/s	34 %
Random 42	3	0.39	0.30	129	1	3.9 MB/s	4.8 MB/s	23 %
Random 43	42	0.84	0.65	158	2	0.6 MB/s	0.6 MB/s	7 %
Random 44	17	0.74	0.05	131	1	0.7 MB/s	0.8 MB/s	12 %
Random 45	27	0.80	0.81	34	1	0.7 MB/s	0.6 MB/s	17 %
Random 46	8	0.16	0.33	158	2	1.6 MB/s	2.1 MB/s	29 %
Random 47	35	0.67	0.83	158	2	0.7 MB/s	0.6 MB/s	19 %
Random 48	22	0.63	0.66	112	1	1.1 MB/s	1.1 MB/s	5 %
Random 49	38	0.70	0.63	62	2	0.6 MB/s	0.5 MB/s	3 %
Random 50	24	0.24	0.43	75	1	0.8 MB/s	0.8 MB/s	4 %
Random 51	12	0.86	0.19	29	2	1.0 MB/s	1.2 MB/s	23 %
Random 52	33	0.33	0.08	116	1	0.6 MB/s	0.7 MB/s	8 %
Random 53	33	0.25	0.81	6	2	0.3 MB/s	0.3 MB/s	11 %
Random 54	7	0.21	0.70	22	2	3.7 MB/s	7.4 MB/s	102 %
Random 55	36	0.32	0.52	119	2	0.8 MB/s	0.5 MB/s	36 %
Random 56	5	0.41	0.27	30	1	3.8 MB/s	3.8 MB/s	0 %
Random 57	21	0.67	0.31	146	1	1.0 MB/s	1.1 MB/s	9 %
Random 58	36	0.23	0.00	134	1	0.6 MB/s	0.6 MB/s	9 %
Random 59	11	0.51	0.32	134	2	1.2 MB/s	1.3 MB/s	7 %
Random 60	26	0.01	0.38	129	2	0.7 MB/s	0.7 MB/s	1 %
Random 61	39	0.15	0.54	97	1	0.6 MB/s	0.5 MB/s	15 %
Random 62	44	0.75	0.38	46	1	0.5 MB/s	0.5 MB/s	3 %
Random 63	32	0.65	0.98	39	2	0.8 MB/s	0.5 MB/s	36 %
Random 64	26	0.02	0.73	45	1	0.7 MB/s	0.6 MB/s	9 %
Random 65	43	0.34	0.84	117	2	0.5 MB/s	0.4 MB/s	21 %
Random 66	37	0.04	0.03	127	2	0.6 MB/s	0.6 MB/s	3 %
Random 67	41	0.26	0.38	60	1	0.5 MB/s	0.5 MB/s	4 %
Random 68	31	0.84	0.24	20	1	0.6 MB/s	0.6 MB/s	6 %
Random 69	46	0.08	0.60	137	1	0.5 MB/s	0.5 MB/s	12 %
Random 70	37	0.84	0.37	102	2	0.6 MB/s	0.6 MB/s	9 %
Random 71	29	0.69	0.85	92	1	0.9 MB/s	0.7 MB/s	24 %
Random 72	13	0.22	0.58	41	2	2.1 MB/s	2.2 MB/s	1 %
Random 73	18	0.49	0.72	93	2	2.4 MB/s	2.4 MB/s	0 %
Random 74	40	0.56	0.95	47	1	0.7 MB/s	0.4 MB/s	37 %
Random 75	24	0.59	0.20	32	1	0.8 MB/s	0.7 MB/s	14 %
Random 76	41	0.43	0.69	88	1	0.6 MB/s	0.5 MB/s	11 %
Random 77	4	0.26	0.29	146	2	3.9 MB/s	4.7 MB/s	20 %

Data for DECstation 5000/200 (Figure 5.5) — Part 3

Workload	un	seq	read	sz	proc	Measured	Predicted	% Err
Random 78	12	0.79	0.83	61	2	4.2 MB/s	3.8 MB/s	10 %
Random 79	17	0.30	0.32	4	2	1.1 MB/s	1.0 MB/s	10 %
Random 80	37	0.26	0.62	72	2	0.5 MB/s	0.5 MB/s	7 %
Random 81	10	0.39	0.62	74	1	2.5 MB/s	2.7 MB/s	8 %
Random 82	32	0.40	0.68	88	2	0.7 MB/s	0.6 MB/s	11 %
Random 83	46	0.01	0.67	72	2	0.4 MB/s	0.5 MB/s	2 %
Random 84	35	0.51	1.00	132	2	0.7 MB/s	0.5 MB/s	31 %
Random 85	37	0.46	0.17	123	2	0.5 MB/s	0.6 MB/s	10 %
Random 86	28	0.14	0.71	31	2	0.6 MB/s	0.5 MB/s	20 %
Random 87	9	0.23	0.92	49	2	4.6 MB/s	6.3 MB/s	36 %
Random 88	15	0.79	0.48	48	2	1.5 MB/s	1.7 MB/s	13 %
Random 89	45	0.34	0.38	12	1	0.4 MB/s	0.3 MB/s	19 %
Random 90	39	0.26	0.05	99	1	0.6 MB/s	0.6 MB/s	2 %
Random 91	45	0.62	0.77	150	2	0.5 MB/s	0.5 MB/s	8 %
Random 92	5	0.50	0.68	132	2	5.4 MB/s	4.4 MB/s	18 %
Random 93	16	0.81	0.36	16	1	1.4 MB/s	1.3 MB/s	2 %
Random 94	13	0.21	0.96	130	2	5.7 MB/s	5.3 MB/s	7 %
Random 95	20	0.21	0.48	36	1	1.4 MB/s	1.5 MB/s	12 %
Random 96	36	0.65	0.83	10	1	0.4 MB/s	0.3 MB/s	7 %
Random 97	21	0.18	0.07	116	2	0.7 MB/s	0.7 MB/s	1 %
Random 98	27	0.74	0.25	9	2	0.6 MB/s	0.5 MB/s	26 %
Random 99	46	0.73	0.26	116	2	0.5 MB/s	0.6 MB/s	4 %

Table A.2: Raw data for DECstation 5000/200 scatter plot. This table gives the data used to plot the current benchmark prediction points in Figures 5.5. Most points of high error have unique-Bytes between 5 and 10 MB.

Data for HP 730 (Figure 5.7) — Part 1

Workload	un	seq	read	sz	proc	Measured	Predicted	% Err
Andrew	4.7	.77	.54	3	1	0.47 MB/s	0.36 MB/s	25 %
Bonnie	100	1.0	0.0	8	1	0.32 MB/s	0.22 MB/s	34 %
IOStone	1.0	.02	.67	2	1	8.6 MB/s	8.0 MB/s	6 %
Sdet	8.0	.48	.56	2	4	0.17 MB/s	0.18 MB/s	9 %
Random 0	9	0.04	0.53	339	1	1.6 MB/s	1.4 MB/s	14 %
Random 1	12	0.50	0.13	128	2	1.2 MB/s	1.2 MB/s	0 %
Random 2	10	0.46	0.99	182	2	1.5 MB/s	2.4 MB/s	58 %
Random 3	1	0.30	0.30	225	1	23.2 MB/s	24.5 MB/s	6 %
Random 4	10	0.73	0.73	129	2	1.4 MB/s	1.6 MB/s	17 %
Random 5	8	0.77	0.37	54	1	1.1 MB/s	1.2 MB/s	5 %
Random 6	3	0.72	0.34	277	1	3.2 MB/s	13.3 MB/s	322 %
Random 7	8	0.73	0.18	262	2	1.3 MB/s	1.8 MB/s	43 %
Random 8	3	0.02	0.37	26	2	11.9 MB/s	8.9 MB/s	26 %
Random 9	1	0.10	0.25	147	2	22.6 MB/s	24.2 MB/s	7 %
Random 10	10	0.32	0.47	62	1	1.0 MB/s	1.0 MB/s	4 %
Random 11	12	0.14	0.74	98	1	1.1 MB/s	1.1 MB/s	1 %
Random 12	3	0.44	0.53	84	2	8.5 MB/s	13.0 MB/s	53 %

Data for HP 730 (Figure 5.7) — Part 2

Workload	un	seq	read	sz	proc	Measured	Predicted	% Err
Random 13	5	0.36	0.54	71	1	1.3 MB/s	1.5 MB/s	10 %
Random 14	5	0.81	0.48	331	1	1.6 MB/s	2.4 MB/s	48 %
Random 15	2	0.38	0.56	233	1	20.9 MB/s	24.9 MB/s	19 %
Random 16	4	0.32	0.56	193	1	2.1 MB/s	2.4 MB/s	15 %
Random 17	6	0.20	0.89	146	2	1.8 MB/s	2.1 MB/s	14 %
Random 18	4	0.56	0.80	41	1	1.8 MB/s	1.7 MB/s	3 %
Random 19	7	0.49	0.68	147	1	1.7 MB/s	1.8 MB/s	2 %
Random 20	4	0.61	0.44	118	2	2.2 MB/s	2.3 MB/s	8 %
Random 21	9	0.36	0.81	356	2	1.7 MB/s	1.7 MB/s	1 %
Random 22	6	0.29	0.50	198	2	1.6 MB/s	1.7 MB/s	4 %
Random 23	1	0.20	0.50	48	1	30.1 MB/s	29.9 MB/s	1 %
Random 24	11	0.65	0.03	253	2	1.2 MB/s	1.5 MB/s	21 %
Random 25	10	0.36	0.99	290	2	1.6 MB/s	2.4 MB/s	55 %
Random 26	6	0.25	0.45	344	1	1.7 MB/s	1.7 MB/s	0 %
Random 27	4	0.50	0.15	337	1	2.9 MB/s	10.8 MB/s	275 %
Random 28	3	0.74	0.74	276	2	10.4 MB/s	19.6 MB/s	88 %
Random 29	9	0.32	0.51	40	1	0.9 MB/s	0.8 MB/s	2 %
Random 30	2	0.79	0.19	132	2	22.8 MB/s	24.3 MB/s	7 %
Random 31	7	0.50	0.04	354	2	1.3 MB/s	1.7 MB/s	24 %
Random 32	9	0.66	0.97	139	1	1.9 MB/s	2.5 MB/s	31 %
Random 33	9	0.49	0.17	264	1	1.5 MB/s	1.6 MB/s	7 %
Random 34	10	0.07	0.80	97	1	1.2 MB/s	1.2 MB/s	1 %
Random 35	9	0.60	0.01	16	2	0.6 MB/s	0.6 MB/s	4 %
Random 36	6	0.64	0.40	226	1	1.8 MB/s	2.1 MB/s	16 %
Random 37	9	0.87	0.94	184	2	1.6 MB/s	2.7 MB/s	66 %
Random 38	5	0.54	0.39	259	2	2.0 MB/s	2.1 MB/s	9 %
Random 39	2	0.02	0.78	295	2	20.4 MB/s	22.7 MB/s	11 %
Random 40	3	0.37	0.08	219	2	20.0 MB/s	19.0 MB/s	5 %
Random 41	5	0.63	0.73	15	1	0.9 MB/s	0.8 MB/s	9 %
Random 42	3	0.31	0.90	177	2	6.5 MB/s	14.5 MB/s	124 %
Random 43	4	0.44	0.39	231	1	2.3 MB/s	2.7 MB/s	17 %
Random 44	2	0.47	0.90	322	1	20.1 MB/s	23.0 MB/s	14 %
Random 45	1	0.30	0.60	19	1	25.6 MB/s	26.0 MB/s	2 %
Random 46	3	0.66	0.55	346	1	18.0 MB/s	21.0 MB/s	16 %
Random 47	9	0.19	0.10	347	2	1.4 MB/s	1.4 MB/s	2 %
Random 48	4	0.62	0.24	324	1	2.1 MB/s	2.5 MB/s	19 %
Random 49	5	0.75	0.11	143	2	2.0 MB/s	2.3 MB/s	16 %
Random 50	11	0.06	0.43	356	1	1.5 MB/s	1.3 MB/s	16 %
Random 51	6	0.29	0.53	52	1	1.2 MB/s	1.1 MB/s	1 %
Random 52	3	0.41	0.47	144	2	21.8 MB/s	24.9 MB/s	14 %
Random 53	11	0.02	0.00	172	1	1.3 MB/s	1.3 MB/s	2 %
Random 54	9	0.67	0.97	104	1	2.1 MB/s	2.4 MB/s	16 %
Random 55	9	0.76	0.88	208	2	1.7 MB/s	2.2 MB/s	28 %
Random 56	3	0.12	0.81	110	1	27.9 MB/s	15.0 MB/s	46 %
Random 57	5	0.01	0.54	333	2	2.2 MB/s	1.7 MB/s	22 %
Random 58	4	0.71	0.90	147	1	3.4 MB/s	3.5 MB/s	2 %
Random 59	6	0.15	0.54	91	2	1.3 MB/s	1.3 MB/s	2 %
Random 60	9	0.40	0.95	62	2	1.1 MB/s	1.5 MB/s	40 %
Random 61	10	0.77	0.62	220	1	1.4 MB/s	1.9 MB/s	32 %

Data for HP 730 (Figure 5.7) — Part 3

Workload	un	seq	read	sz	proc	Measured	Predicted	% Err
Random 62	4	0.27	0.20	200	1	2.4 MB/s	2.7 MB/s	10 %
Random 63	10	0.04	0.92	83	1	1.3 MB/s	1.4 MB/s	10 %
Random 64	9	0.34	0.17	149	2	1.3 MB/s	1.3 MB/s	5 %
Random 65	9	0.85	0.95	270	1	1.9 MB/s	3.0 MB/s	58 %
Random 66	11	0.73	0.21	176	1	0.9 MB/s	1.6 MB/s	70 %
Random 67	7	0.39	0.72	189	1	1.7 MB/s	1.9 MB/s	10 %
Random 68	2	0.28	0.02	363	1	22.1 MB/s	21.2 MB/s	4 %
Random 69	8	0.80	0.82	216	2	1.2 MB/s	2.2 MB/s	80 %
Random 70	8	0.13	0.12	336	1	1.2 MB/s	1.5 MB/s	22 %
Random 71	7	0.05	0.53	302	1	1.8 MB/s	1.6 MB/s	15 %
Random 72	1	0.51	0.76	261	2	19.2 MB/s	24.2 MB/s	25 %
Random 73	7	0.19	0.99	262	1	2.3 MB/s	2.8 MB/s	18 %
Random 74	11	0.16	0.94	23	1	0.7 MB/s	0.8 MB/s	12 %
Random 75	2	0.23	0.28	159	1	23.8 MB/s	24.5 MB/s	3 %
Random 76	7	0.40	0.59	186	2	1.5 MB/s	1.6 MB/s	8 %
Random 77	11	0.24	0.57	119	2	1.2 MB/s	1.1 MB/s	7 %
Random 78	1	0.16	0.69	250	1	22.3 MB/s	24.8 MB/s	11 %
Random 79	2	0.78	0.05	105	1	26.7 MB/s	26.8 MB/s	0 %
Random 80	9	0.79	0.20	124	1	1.5 MB/s	1.6 MB/s	10 %
Random 81	11	0.15	0.53	30	2	0.4 MB/s	0.6 MB/s	49 %
Random 82	4	0.84	0.85	273	1	2.1 MB/s	3.6 MB/s	70 %
Random 83	3	0.35	0.45	262	2	4.8 MB/s	13.5 MB/s	180 %
Random 84	11	0.22	0.37	154	2	1.0 MB/s	1.2 MB/s	18 %
Random 85	8	0.75	0.55	145	2	1.5 MB/s	1.7 MB/s	14 %
Random 86	8	0.42	0.43	361	2	1.4 MB/s	1.5 MB/s	8 %
Random 87	8	0.17	0.85	246	1	1.7 MB/s	1.9 MB/s	12 %
Random 88	6	0.66	0.42	237	1	1.8 MB/s	2.1 MB/s	18 %
Random 89	10	0.12	0.16	212	1	1.3 MB/s	1.4 MB/s	14 %
Random 90	2	0.60	0.26	195	2	18.7 MB/s	24.3 MB/s	30 %
Random 91	8	0.71	0.88	137	2	1.5 MB/s	2.0 MB/s	35 %
Random 92	8	0.53	0.52	128	2	1.3 MB/s	1.4 MB/s	11 %
Random 93	11	0.26	0.30	93	1	1.0 MB/s	1.1 MB/s	11 %
Random 94	4	0.06	0.65	209	2	5.5 MB/s	9.3 MB/s	68 %
Random 95	3	0.53	0.71	227	2	17.6 MB/s	13.2 MB/s	25 %
Random 96	3	0.47	0.47	7	2	7.1 MB/s	5.1 MB/s	28 %
Random 97	3	0.14	0.31	7	1	15.4 MB/s	8.5 MB/s	45 %
Random 98	11	0.10	0.92	335	2	1.4 MB/s	1.7 MB/s	28 %
Random 99	2	0.72	0.33	14	1	22.9 MB/s	22.4 MB/s	2 %

Table A.3: Raw data for HP 730 scatter plot. This table gives the data used to plot the current benchmark prediction points in Figures 5.7. Most points of high error have uniqueBytes between 3 and 4 MB.



Data for Convex C240 (Figure 5.9) — Part 1

Workload	un	seq	read	sz	proc	Measured	Predicted	% Err
Andrew	4.7	.77	.54	3	1	2.9 MB/s	2.9 MB/s	2 %
Bonnie	100	1.0	0.0	8	1	2.0 MB/s	1.4 MB/s	31 %
IOStone	1.0	.02	.67	2	1	2.4 MB/s	2.1 MB/s	14 %
Sdet	8.0	.48	.56	2	4	5.7 MB/s	2.3 MB/s	60 %
Random 0	1077	0.80	0.41	339	2	10.7 MB/s	12.3 MB/s	16 %
Random 1	768	0.85	0.57	341	1	14.1 MB/s	17.3 MB/s	23 %
Random 2	1121	0.85	0.37	221	2	9.8 MB/s	10.1 MB/s	4 %
Random 3	1282	0.08	0.30	1001	2	8.9 MB/s	5.7 MB/s	36 %
Random 4	266	0.22	0.97	370	2	22.3 MB/s	23.9 MB/s	7 %
Random 5	1591	0.11	0.46	571	1	6.8 MB/s	5.3 MB/s	21 %
Random 6	313	0.21	0.26	654	2	7.0 MB/s	7.0 MB/s	1 %
Random 7	209	0.61	0.01	530	1	5.4 MB/s	7.7 MB/s	43 %
Random 8	901	0.52	0.91	870	2	15.8 MB/s	18.4 MB/s	16 %
Random 9	756	0.39	0.65	236	1	13.6 MB/s	13.3 MB/s	2 %
Random 10	972	0.57	0.77	729	1	10.8 MB/s	13.3 MB/s	23 %
Random 11	709	0.25	0.69	83	2	10.4 MB/s	9.4 MB/s	9 %
Random 12	374	0.61	0.34	915	1	6.9 MB/s	7.0 MB/s	2 %
Random 13	571	0.53	0.17	456	2	8.7 MB/s	9.2 MB/s	6 %
Random 14	983	0.70	0.06	473	2	10.8 MB/s	7.5 MB/s	30 %
Random 15	1118	0.03	0.55	259	1	6.6 MB/s	6.4 MB/s	2 %
Random 16	269	0.39	0.15	247	2	7.2 MB/s	7.5 MB/s	4 %
Random 17	539	0.41	0.59	1010	1	9.0 MB/s	7.6 MB/s	16 %
Random 18	728	0.34	0.63	452	1	11.9 MB/s	13.6 MB/s	14 %
Random 19	853	0.41	0.29	466	2	10.8 MB/s	9.3 MB/s	14 %
Random 20	1438	0.87	0.39	121	2	6.1 MB/s	6.1 MB/s	0 %
Random 21	232	0.59	0.33	409	2	8.3 MB/s	9.6 MB/s	16 %
Random 22	760	0.79	0.38	753	1	10.8 MB/s	10.7 MB/s	1 %
Random 23	445	0.28	0.44	713	2	9.3 MB/s	7.5 MB/s	19 %
Random 24	540	0.81	0.76	132	2	14.5 MB/s	18.1 MB/s	25 %
Random 25	1593	0.31	0.67	206	1	5.1 MB/s	5.3 MB/s	3 %
Random 26	706	0.24	0.93	839	2	13.6 MB/s	19.0 MB/s	39 %
Random 27	1405	0.03	0.69	544	2	8.0 MB/s	6.9 MB/s	14 %
Random 28	598	0.43	0.32	85	1	5.6 MB/s	5.8 MB/s	3 %
Random 29	1004	0.53	0.15	622	2	10.4 MB/s	6.3 MB/s	40 %
Random 30	1378	0.14	0.82	10	1	1.0 MB/s	1.2 MB/s	15 %
Random 31	352	0.79	0.67	717	1	7.6 MB/s	16.4 MB/s	116 %
Random 32	1197	0.06	0.62	409	2	8.4 MB/s	7.1 MB/s	16 %
Random 33	1257	0.54	0.66	270	2	8.0 MB/s	8.7 MB/s	9 %
Random 34	151	0.82	0.27	508	1	6.9 MB/s	12.0 MB/s	74 %
Random 35	732	0.59	0.87	523	2	19.3 MB/s	25.7 MB/s	33 %
Random 36	371	0.00	0.38	226	1	7.1 MB/s	7.4 MB/s	3 %
Random 37	386	0.17	0.17	631	2	7.6 MB/s	6.9 MB/s	9 %
Random 38	1468	0.78	0.22	986	1	8.5 MB/s	9.0 MB/s	6 %
Random 39	177	0.12	0.34	869	1	4.0 MB/s	5.7 MB/s	40 %
Random 40	850	0.69	0.82	1012	2	12.3 MB/s	15.6 MB/s	27 %
Random 41	1379	0.43	0.98	882	2	10.3 MB/s	15.5 MB/s	50 %
Random 42	362	0.06	0.53	13	1	2.0 MB/s	2.1 MB/s	8 %
Random 43	154	0.39	0.30	823	1	4.7 MB/s	6.5 MB/s	40 %
Random 44	1427	0.84	0.65	1008	2	12.0 MB/s	13.0 MB/s	9 %

Data for Convex C240 (Figure 5.9) — Part 2

Workload	un	seq	read	sz	proc	Measured	Predicted	% Err
Random 45	645	0.74	0.05	833	1	9.0 MB/s	7.7 MB/s	15 %
Random 46	976	0.80	0.81	198	1	14.4 MB/s	17.8 MB/s	23 %
Random 47	343	0.16	0.33	1010	2	7.5 MB/s	5.3 MB/s	30 %
Random 48	1219	0.67	0.83	1010	2	11.6 MB/s	14.9 MB/s	29 %
Random 49	811	0.63	0.66	707	1	12.3 MB/s	14.2 MB/s	15 %
Random 50	1319	0.70	0.63	385	2	9.6 MB/s	10.0 MB/s	4 %
Random 51	860	0.24	0.43	471	1	10.9 MB/s	9.1 MB/s	17 %
Random 52	441	0.86	0.19	169	2	9.0 MB/s	9.7 MB/s	7 %
Random 53	1172	0.33	0.08	735	1	7.6 MB/s	6.7 MB/s	12 %
Random 54	1179	0.25	0.81	17	2	1.8 MB/s	1.9 MB/s	10 %
Random 55	314	0.21	0.70	120	2	11.7 MB/s	10.8 MB/s	8 %
Random 56	1283	0.32	0.52	758	2	9.0 MB/s	7.5 MB/s	16 %
Random 57	241	0.41	0.27	173	1	7.2 MB/s	6.8 MB/s	5 %
Random 58	754	0.67	0.31	937	1	9.9 MB/s	7.4 MB/s	26 %
Random 59	1281	0.23	0.00	855	1	7.3 MB/s	5.4 MB/s	26 %
Random 60	433	0.51	0.32	855	2	9.0 MB/s	6.7 MB/s	25 %
Random 61	952	0.01	0.38	824	2	10.8 MB/s	5.5 MB/s	49 %
Random 62	1387	0.15	0.54	611	1	7.0 MB/s	6.8 MB/s	3 %
Random 63	1533	0.75	0.38	280	1	7.2 MB/s	7.5 MB/s	5 %
Random 64	1133	0.65	0.98	235	2	14.4 MB/s	17.8 MB/s	24 %
Random 65	945	0.02	0.73	269	1	11.9 MB/s	12.1 MB/s	1 %
Random 66	1502	0.34	0.84	741	2	10.1 MB/s	9.7 MB/s	4 %
Random 67	1297	0.04	0.03	807	2	7.5 MB/s	4.7 MB/s	38 %
Random 68	1436	0.26	0.38	371	1	5.5 MB/s	5.6 MB/s	2 %
Random 69	1105	0.84	0.24	105	1	5.3 MB/s	6.6 MB/s	26 %
Random 70	1603	0.08	0.60	872	1	7.1 MB/s	6.0 MB/s	16 %
Random 71	1291	0.84	0.37	646	2	10.0 MB/s	11.0 MB/s	10 %
Random 72	1048	0.69	0.85	577	1	9.7 MB/s	19.3 MB/s	100 %
Random 73	515	0.22	0.58	247	2	10.8 MB/s	10.2 MB/s	6 %
Random 74	665	0.49	0.72	588	2	14.2 MB/s	17.5 MB/s	23 %
Random 75	1410	0.56	0.95	288	1	11.3 MB/s	13.7 MB/s	21 %
Random 76	885	0.59	0.20	188	1	9.0 MB/s	8.1 MB/s	10 %
Random 77	1446	0.43	0.69	552	1	8.3 MB/s	8.8 MB/s	6 %
Random 78	189	0.26	0.29	938	2	4.9 MB/s	5.2 MB/s	6 %
Random 79	457	0.79	0.83	375	2	18.4 MB/s	24.7 MB/s	34 %
Random 80	653	0.30	0.32	4	2	0.9 MB/s	1.1 MB/s	21 %
Random 81	1292	0.26	0.62	452	2	8.4 MB/s	7.6 MB/s	9 %
Random 82	414	0.39	0.62	463	1	10.8 MB/s	12.2 MB/s	13 %
Random 83	1120	0.40	0.68	551	2	10.2 MB/s	11.0 MB/s	8 %
Random 84	1604	0.01	0.67	446	2	7.1 MB/s	5.6 MB/s	21 %
Random 85	1245	0.51	1.00	838	2	12.0 MB/s	18.3 MB/s	52 %
Random 86	1303	0.46	0.17	784	2	9.4 MB/s	6.7 MB/s	29 %
Random 87	990	0.14	0.71	179	2	9.8 MB/s	9.8 MB/s	0 %
Random 88	386	0.23	0.92	298	2	19.4 MB/s	21.1 MB/s	9 %
Random 89	554	0.79	0.48	291	2	12.6 MB/s	14.1 MB/s	12 %
Random 90	1570	0.34	0.38	58	1	2.0 MB/s	2.3 MB/s	12 %
Random 91	1366	0.26	0.05	627	1	6.4 MB/s	5.4 MB/s	15 %
Random 92	1555	0.62	0.77	955	2	10.8 MB/s	10.8 MB/s	0 %
Random 93	214	0.50	0.68	844	2	5.4 MB/s	11.3 MB/s	109 %

Data for Convex C240 (Figure 5.9) — Part 3

Workload	un	seq	read	sz	proc	Measured	Predicted	% Err
Random 94	594	0.81	0.36	81	1	8.3 MB/s	7.9 MB/s	5 %
Random 95	482	0.21	0.96	827	2	11.0 MB/s	18.2 MB/s	65 %
Random 96	748	0.21	0.48	216	1	9.7 MB/s	8.8 MB/s	9 %
Random 97	1272	0.65	0.83	43	1	5.1 MB/s	4.2 MB/s	16 %
Random 98	778	0.17	0.07	739	2	9.4 MB/s	6.1 MB/s	36 %
Random 99	977	0.74	0.25	36	2	3.7 MB/s	3.3 MB/s	9 %

Table A.4: Raw data for Convex C240 scatter plot. This table gives the data used to plot the current benchmark prediction points in Figures 5.9. Most points of high error have sizeMean over 300 KB.

System	Workload	Measured	Predicted	% Error
SPARCstation 1+	workstation	1.8 MB/s	1.7 MB/s	6 %
	large_utility	2.1 MB/s	2.0 MB/s	2 %
	scientific_write	0.46 MB/s	0.36 MB/s	20 %
	scientific_read	0.59 MB/s	0.54 MB/s	9 %
	database	0.10 MB/s	0.15 MB/s	44 %
DECstation 5000/200	workstation	3.8 MB/s	3.2 MB/s	15 %
	large_utility	2.5 MB/s	2.0 MB/s	17 %
	scientific_write	0.41 MB/s	0.46 MB/s	10 %
	scientific_read	0.40 MB/s	0.38 MB/s	5 %
	database	0.30 MB/s	0.33 MB/s	7 %
HP 730	workstation	11.5 MB/s	9.8 MB/s	15 %
	large_utility	0.54 MB/s	0.47 MB/s	13 %
	scientific_write	0.53 MB/s	0.77 MB/s	45 %
	scientific_read	1.2 MB/s	1.2 MB/s	6 %
	database	0.13 MB/s	0.14 MB/s	5 %
Convex C240	workstation	4.2 MB/s	3.7 MB/s	12 %
	large_utility	6.5 MB/s	4.2 MB/s	36 %
	scientific_write	6.1 MB/s	6.4 MB/s	4 %
	scientific_read	15.5 MB/s	17.2 MB/s	11 %
	database	0.75 MB/s	0.73 MB/s	2 %

Table A.5: Raw data for ratio prediction. This table gives the raw data used in calculating the ratios of Figure 5.13.

---

## Appendix B

### Prediction Algorithm

---

This chapter describes an algorithm for predicting performance based on the families of graphs reported by the self-scaling benchmark.

This algorithm takes as input two items (all variables are in **bold**).

- The graphs from the self-scaling benchmark, that is, measured workloads for a number of focal points { focal point #0 to focal point #(focalNum-1) } each with a different focal point for uniqueBytes **focalPoint.uniqueBytes[i]**, but otherwise the same focal point **focal.sizeMean, focal.readFrac, focal.seqFrac, focal.processNum**.
- An I/O workload { **target.uniqueBytes, target.sizeMean, target.readFrac, target.seqFrac, target.processNum** } for which to predict performance.

This algorithm returns an estimate of the performance for the input workload.

- (1) Choose the focal point upon which to predict performance for the target workload. Pick the focal point whose value of uniqueBytes has performance closest to the performance of target.uniqueBytes.

```
/* find the two focal points whose values of uniqueBytes
   form the tightest range which includes target.uniqueBytes */

focalLess = focal point with largest value of uniqueBytes
            less than target.uniqueBytes

focalMore = focal point with smallest value of uniqueBytes
            greater than target.uniqueBytes

/* pick the focal point whose value of uniqueBytes has
   performance closest to the performance of target.uniqueBytes
   */

/* performanceLess is the performance at focal point focalLess */
performanceLess = performance of workload
                 (focal.uniqueBytes[focalLess], focal.sizeMean,
                  focal.readFrac, focal.seqFrac, focal.processNum)

/* performanceMore is the performance at focal point focalMore
   */

performanceMore = performance of workload
                 (focal.uniqueBytes[focalMore], focal.sizeMean,
                  focal.readFrac, focal.seqFrac, focal.processNum)

/* performance is the performance of the workload which is
   the same as the focal point except for uniqueBytes,
   which is equal to target.uniqueBytes. */
performance = performance of workload (target.uniqueBytes,
                                       focal.sizeMean, focal.readFrac, focal.seqFrac, focal.processNum)

if (performance is closer to performanceLess) {
    focalUse = focalLess
} else {
    focalUse = focalMore
}

performance = performance at focal point focalUse
```

- (2) Use focal point `focalUse` to adjust performance for differences between the focal point and the target workload, one parameter at a time.

```
for each parameter (uniqueBytes, sizeMean, readFrac, seqFrac,
                    processNum)
  /* adjust for parameter */

  /* performance1 is the performance at focal point focalUse
   */

  performance1 = performance of workload
    (focal.uniqueBytes[focalUse], focal.sizeMean, focal.readFrac,
     focal.seqFrac, focal.processNum)

  /* performance2 is the performance of the workload which
   is the same as the focal point except for parameter,
   which has the value of the workload whose perfor-
   mance is being predicted. Here I've shown the
   workload assuming parameter is sizeMean */

  performance2 = performance of workload
    (focal.uniqueBytes[focalUse], target.sizeMean, focal.readFrac,
     focal.seqFrac, focal.processNum)

  performance *= performance2 / performance1
end for

return (performance)
```

---

## Appendix C

### Proof of Performance Equation

---

This appendix gives a proof that the overall performance equation must take the form of a product of independent functions, where each function is a function of exactly one workload parameter (see Figure 5.1 on page 83).

I first show this is true with two workload parameters A and B.

**Given:** Performance equation  $f(A, B)$

$$\text{and } \frac{f(A, B_1)}{f(A, B_2)} = \text{constant}$$

**Show:**  $f(A, B) = g(A) \times h(B)$

**Proof:**

$$\frac{f(A,B)}{f(A,B_1)} = \text{constant (for all A with B fixed)}$$

$$\frac{f(A,B)}{f(A,B_1)} = \frac{f(A_1,B)}{f(A_1,B_1)}$$

$$f(A,B) = \frac{f(A_1,B)}{f(A_1,B_1)} \times f(A,B_1)$$

$$f(A,B) = g(B) \times h(A)$$

$$\text{where } g(B) = \frac{f(A_1,B)}{f(A_1,B_1)}$$

$$\text{and } h(A) = f(A,B_1)$$

*Q.E.D.*

Next, I generalize this to multiple workload parameters A, B, C, D...

$$\frac{f(A,B,C,D,\dots)}{f(A,B_1,C,D,\dots)} = \frac{f(A_1,B,C,D,\dots)}{f(A_1,B_1,C,D,\dots)}$$

$$f(A,B,C,D,\dots) = \frac{f(A_1,B,C,D,\dots)}{f(A_1,B_1,C,D,\dots)} \times f(A,B_1,C,D,\dots)$$

This equation has the form of a product of functions, with each each term having at least one less independently-varying parameter. By recursing down to two workload parameters and using the commutative property of multiplication/division, the overall performance equation will reduce to a product of functions, with each function being a function of one workload parameter.

*Q.E.D.*



