On modeling the relative fitness of storage

Michael P. Mesnier

December 19, 2007
CMU-PDL-08-107

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis committee

Prof. Gregory R. Ganger, Chair
Prof. Christos Faloutsos
Dr. Mic Bowman (Intel Research)
Dr. Arif Merchant (Hewlett-Packard Laboratories)

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

I dedicate this work to my family and friends.

Acknowledgements

I would first like to thank my advisor, Greg Ganger, for the opportunity to work in the Parallel Data Lab with such an amazing group of individuals. The raw talent and enthusiasm are what make the PDL truly unique, and I feel honored to have been part of it. I would also like to thank the other members of my thesis committee, including Dr. Mic Bowman (Intel), Dr. Arif Merchant (HP), and Professor Christos Faloutsos (CMU) — many thanks to both Mic and Arif for making the long trips to Pittsburgh, and to Christos for our numerous meetings. Each of these discussions helped shape this work.

I had the pleasure of working with many graduate students while at CMU, and I would like to thank them all. This includes the ABLE team (Eno Thereska, and Dan Ellard from Harvard), the //TRACE team (James Hendricks, Julio Lopez, Raja Sambasivan, and Matthew Wachs), and those that participated in the relative fitness discussions (Brandon Salmon, Raja Sambasivan, Matthew Wachs, and post. doc. Alice Zheng). For the many lively storage discussions, I also thank Steve Schlosser and Lily Mummert from Intel Research — I'll never look at sushi the same way again! And to the wonderful support staff, including Karen Lindenfelser, Helen Conti, Shellee Lank (Intel Research), and Michael Stroucken, I hope you all know how much I appreciated having you there.

Of course, many individuals helped get me to this point, including Dr. John Erhart (my undergraduate advisor), Dr. Jorge More (my supervisor at Argonne National Laboratory); and Jerry Baugh, Beth Reinders, and Don Cameron (my first managers at Intel). Their many words of encouragement mean more than they know. I would like to especially thank Don Cameron for making my CMU "assignment" possible, as well as my subsequent Intel managers (Raj Ramanujan, Limor Fix, and Todd Mowry) for seeing me through the many Intel "reorgs." This would not have been possible without their support.

Finally, to those that made my stay in Pittsburgh and enjoyable one — thank you. This includes the late night runs to Steak n' Shake, D&B, sailing on Lake Arthur, sailing on the Monongahela (Yes, it can be done — but watch out for the barges!), the many impromptu "field trips," BLTs, regularly catching the 29/30, and helping me develop my appreciation for single malt scotch. I hope we all have the pleasure of meeting again someday.

Abstract

Storage management is usually handled by skilled system administrators. The specific task of configuring and allocating disk space for applications, often referred to as storage system design, is especially time-consuming and error-prone. Automated storage system design, a solution proposed by many, relies on fast and accurate performance predictions. However, challenges with conventional performance modeling have prevented such automation from being fully realized in practice.

Relative fitness is a new approach to modeling the performance of storage systems. In contrast to conventional models that predict the performance of storage systems based on the characteristics of workloads, referred to in this dissertation as absolute models, relative fitness models predict performance differences as workloads are moved across storage systems. There are two primary advantages. First, because relative fitness models are constructed for each pair of storage systems, the feedback of a closed workload can be captured (e.g., how the I/O arrival rate changes as the workload moves from storage system A to storage system B). Second, relative fitness models allow performance and resource utilization to be used in addition to workload characteristics. This is beneficial when workload characteristics are difficult to obtain or concisely express. For example, rather than trying to describe the spatio-temporal characteristics of a workload, one could use the observed performance and cache hit rate of storage system A to help predict the performance of storage system B.

This dissertation describes the steps necessary to build a relative fitness model, with an approach that is general enough to be used with any black-box modeling technique. Relative fitness models and absolute models are compared across a variety of workloads and disk arrays (RAID). When compared to absolute models, relative fitness models reduce the bandwidth prediction error up to 53%, throughput up to 23%, and latency up to 20%. In general, the best predictors of the relative fitness models are performance observations, followed by conventional workload characteristics.

Relative fitness models can be used in automated storage system design in a similar way that absolute models are used. Specifically, workloads can be observed on the storage systems that they are initially assigned to, relative fitness models can use these observations to predict the performance of different assignments, and optimization techniques can be used to select an assignment that optimizes performance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Storage systems can be costly to manage. The consequences of bad management range from poor performance (violated service-level agreements) to lost data. System administrators try to ensure that the workload characteristics of applications are appropriately matched to the performance and availability characteristics of the storage to which they are assigned. This design process includes the configuration of storage systems (e.g., selecting a RAID level for a disk array) and the allocation of space for applications. In both cases, a performance prediction is desirable (e.g., "is RAID-10 faster than RAID-5?").

Though seasoned administrators may develop intuition as to the best way to manage storage, the many interactions between an application workload and a storage system make accurate performance prediction difficult. Further, because there are multiple options to consider, the problem of finding the "optimal" storage system design (e.g., minimum cost, maximum performance, or best price-performance) is one of bin-packing. Administrators must decide how to pack workloads into storage devices, but finding an optimal solution would require that an exponential number of possibilities be explored. Because this is not humanly possible in most cases, administrators often use rules-of-thumb. As might be expected, this can lead to suboptimal designs and wasted resources. Anyone who has ever attempted to efficiently pack luggage into a car's trunk has experienced the bin-packing problem first-hand. Rules-of-thumb and heuristics can be used to guide the search (e.g., pack the big suitcases first) but can be inefficient (e.g., resulting in unreachable space).

Given the high costs of storage management, there is a big payoff if automated tools can offload, or at least assist in, the design process. In fact, automated storage system design has been an area of interest for well over 30 years. Solutions generally take the form of optimization solvers and performance models that predict the outcome of various designs. Using these models, optimization solvers can search the large solution space for a near-optimal design.

Analytical or statistical models are the type of models typically used in such optimization. Conventionally, these models take as input a concise description of a workload, referred to as workload characteristics. As output, a performance prediction is made. These models assume that a workload is described in an absolute manner, such that its characteristics are independent of the storage system on which they were observed. For example, if a workload is characterized as read-mostly on storage system A, a model of storage system B assumes that this characteristic is not relative to storage system A

and that the workload will also be read-mostly on storage system B. In this dissertation, such models are referred to as *absolute models*.

Practical challenges with workload characterization have limited the use of absolute models in automated storage system design. First, compressing the I/O workload of an application into concise workload characteristics, without losing performance-critical information (e.g., spatial locality and temporal locality), is challenging. Indeed, creating characteristics that are concise is at odds with creating characteristics that are expressive. Some amount of information is usually lost in the compression, and the lost information can affect a model's prediction accuracy. Second, it is often the case that the characteristics of a workload, as observed by a storage system, are not actually absolute. For example, the average I/O request size of a workload can change between two storage systems; one storage system may be slower, allowing an operating system more time to coalesce and aggregate I/O requests, thereby resulting in larger requests.

To specifically address the challenges with absolute modeling, this dissertation presents a new modeling technique called *relative fitness*. A relative fitness model uses the observed performance and resource utilization of a workload on one storage system, in addition to workload characteristics, to predict the performance of another storage system. Performance, resource utilization, and workload characteristics are assumed specific, or relative, to a given storage system, hence the term "relative modeling." In doing so, the dependence on workload characteristics is reduced, due to the use of performance and resource utilization, and changes in workload characteristics can be implicitly modeled. In addition, relative fitness models predict performance ratios, called relative fitness values, rather than performance itself. As an example, a relative fitness model might learn that a sequential, read-only workload with an observed bandwidth of 80 MB/sec and a queue depth of 13 on storage system A will run 30% faster on storage system B.

For every pair of storage systems A and B, two relative fitness models are trained: one to predict the performance of B relative to A and a second to predict the performance of A relative to B. The models are black-box models, thereby requiring no knowledge of each storage system's internals (e.g., number of drives, RAID level, cache sizes and eviction policies, and bus topology). The models learn various relative fitness values by discovering statistical correlations between the observations of one storage system and the performance of another. To make a performance prediction for a new workload, one inputs workload observations into the appropriate relative fitness model and multiplies the predicted relative fitness value (the performance ratio) by the observed performance. Figure 1 illustrates.

1.1 Thesis statement and contributions

Thesis statement: The prediction accuracy of a statistical, black-box performance model can be improved by training it to predict the performance of one storage system relative to the observations of another. Observations include workload characteristics, performance, and resource utilization.

This dissertation presents five independent discoveries. First, the block-level workload characteristics of an application can change as the application is moved across storage systems. Second, the change can impact the accuracy of an absolute model. Third, storage systems can be modeled relative to one another

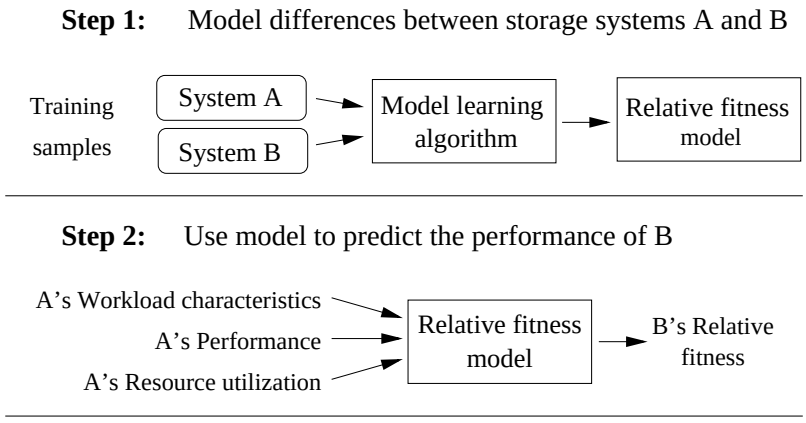

Figure 1.1: Relative fitness models predict changes in performance between two storage systems. Sample workloads are used to train a model to predict how two storage systems will differ for a given workload.

and reduce the effect of changing workloads. Fourth, the observed performance and resource utilization of a workload on one storage system can be used to predict its performance on a different storage system. Finally, performance ratios (relative fitness values) are better predictors of performance than constant performance values. Collectively, these discoveries establish the motivation and theory behind relative fitness modeling.

1.2 Thesis organization

The background material for this dissertation is organized as 4 chapters: automated storage management (Chapter 2), measuring storage performance (Chapter 3), characterizing storage workloads (Chapter 4), and predicting storage performance (Chapter 5). Though not comprehensively, Chapters 2 through 5 introduce many basic concepts behind storage systems and performance modeling.

Chapter 2 begins with basic terminology (e.g., "workload") that will be used throughout this dissertation and provides a more in-depth discussion of automated storage management, specifically storage system design. A hypothetical automated storage system is described with an elaboration of the challenges that have prevented it from becoming fully realized in practice.

The performance metrics (bandwidth, throughput, and latency) that the relative fitness models are trained to predict are presented in Chapter 3. Techniques for measuring performance, and common pitfalls, are discussed. In addition, a variety of industry benchmarks and synthetic workloads are described, some of which are used to evaluate the relative fitness models.

Workload characterization is discussed in Chapter 4. Basic workload characteristics are presented first, including operation mixes, I/O request sizes, spatial access patterns, and levels of concurrency (e.g., queue depths). More expressive characteristics are also presented, including temporal measures like the "burstiness" of a workload, self-similarity in an I/O stream, and correlations between space and time (e.g., certain blocks accessed at certain times). The challenges with workload characterization are discussed in greater detail.

Chapter 5 introduces various predictive techniques. I/O trace replay, synthetic I/O, simulation mod-

els, and emulation models are described, in addition to analytical and statistical models. Particular attention is given to the statistical models, as they form the foundation for relative fitness modeling.

Chapter 6 presents relative fitness modeling. Despite the many differences between a relative fitness model and an absolute model, one can regard relative fitness modeling as an evolution of absolute modeling, and the derivation of relative fitness is presented as such. It is shown how relative fitness models can be implemented using classification and regression trees (CART).

The CART models are evaluated in Chapter 7. Relative fitness modeling is compared with absolute modeling using a variety of application workloads (synthetic I/O, trace replay, Postmark, and TPC-C) and disk arrays. It is shown that the relative fitness models can reduce the median relative error of the bandwidth predictions from 67% to 14%, throughout from 36% to 13%, and latency from 34% to 14%. Furthermore, the relative fitness models reduce error with much simpler models (Occam's razor) that also require less training data.

Chapter 8 summarizes the thesis and provides some thoughts for future work.

Chapter 2

Automated storage management

"Storage management" is a large umbrella that covers a range of activities, including purchasing & installation, storage system design, performance monitoring, tuning, backup & restore, archiving, and system upgrades. Plus, there is infrastructure that must be managed, such as networking, power, and cooling. Further, all of the above must respect data center policies related to quality of service (performance), security & privacy, data retention, and availability (e.g., "What's the disaster recovery plan?"). In short, storage management is not for the faint of heart — especially when a corporation and its customers' data are at stake — and it can consume a significant amount of a system administrator's time.

This chapter introduces storage system design, a component of storage management that can be especially time-consuming and error-prone. A "storage system" typically refers to one or more disk arrays (RAID), and the "design" is how they are each configured and assigned work. This is one of many areas of storage management that can benefit from automation and is the context in which relative fitness models are intended to be used.

Section 2.1 begins with a canonical depiction of a file and storage system, so as to introduce the basic storage concepts and terminology that will be used throughout the rest of this dissertation. Section 2.2 continues with a discussion of the high costs of storage management, a trend in the storage industry that is making the management problem even worse (storage clusters), and the general need for automated management. Section 2.3 focuses the discussion on storage system design and the problem it poses for large data centers. Section 2.4 introduces automated design, progress that has been made in this area, and the challenges going forward. Section 2.5 summarizes the chapter and introduces the potential for relative fitness modeling.

2.1 Basic concepts and terminology

Storage systems aggregate the capabilities of multiple storage devices in order to increase overall storage capacity, reliability, availability, or performance. Examples include disk arrays, tape libraries, disk-to-tape backup systems, and hybrid drives (disk + Flash). Disk arrays, given their prevalence in the data center, are the focus of this dissertation.

Most disk arrays have an administrative interface for creating *logical volumes*. Logical volumes may

span multiple disks by concatenating their block address spaces (linear mode), allocating blocks in a round-robin fashion (striping), or employing some form of data redundancy that may also involve striping (e.g., RAID [42, 29]). It is the job of a system administrator to set the type and size of each volume. File systems, databases, and other applications are created within these logical volumes (e.g., `C:` and `D:` are file systems that may reside on the same disk array, but in separate logical volumes). Blocks within a logical volume are *logical blocks* which map to *physical blocks* in the underlying storage devices. A logical volume is presented to an operating system as though it were a single storage device. Therefore, when accessing data in a logical volume, an OS can use the same read/write block interface that is uses to access a storage device (e.g., the SCSI or ATA block command set).

Figure 2.1 (left half) illustrates a canonical file and storage system. An abstracted view (right half) is the stack that will be referred to throughout the rest of this dissertation. Although only a file system is used in the figure, the stack could have be shown with a database or any other application that directly accesses the storage system. The I/O *workload* of any such application are the logical block I/O requests that it issues into the storage system, and the block I/O of a specific process (or thread) is referred to as a *stream*. As such, an I/O workload is composed of one or more streams.

This work is concerned with predicting the performance of I/O workloads, not individual streams, and it is assumed that such workloads are not sharing the storage system with any other workloads. Appendix B contains an early follow-on to this work that predicts the impact of resource sharing.

When referring to the I/O workload an application (e.g., Postmark [7]) running on top of a mounted file system, this work is referring to its block I/O, after the file requests have passed through the file system and been translated to logical block requests. That is, a workload is described by its observed block-level workload characteristics (e.g., read/write ratio, average request size, spatial randomness) and its observed block-level performance (e.g., bandwidth, throughput, latency). Application-level workload characteristics and performance (i.e., as observed by an application) are not used in this dissertation.

2.2 The high costs of management

At the highest level, the goal of storage management is to build and maintain logical volumes for application servers. However, much work goes behind the simple "`C:`" that is presented to an application.

It is a well-known fact that it costs more to manage storage than it does to buy it [2, 28, 26, 35, 45, 44], with estimates as high as 8:1 (Gartner 2003). That is, it costs $8 to manage every $1 of storage. Although studies vary, the management costs usually include everything in addition to the raw storage (e.g., storage management software and skilled system administrators). Some studies have found that 1 administrator is needed every 1-20 TB of data [26, 44], which is equivalent to 1 administrator every 2–40 disk drives. Combining these alarming ratios with a 50-100% yearly increase in storage capacity, which is not uncommon in the data center [34], one can understand why the total cost of ownership (TCO) of storage is of immediate concern in the enterprise.

In an effort to reduce storage TCO, and in some cases improve scalability, many storage systems have begun to move from monolithic designs (like that shown in Figure 2.1) to clusters [1, 12, 21, 25, 36, 50, 41, 58], where logical volumes may span multiple disk arrays. This is often referred to as cluster-based storage. Indeed, storage systems can benefit from the same advantages that clustering brought to

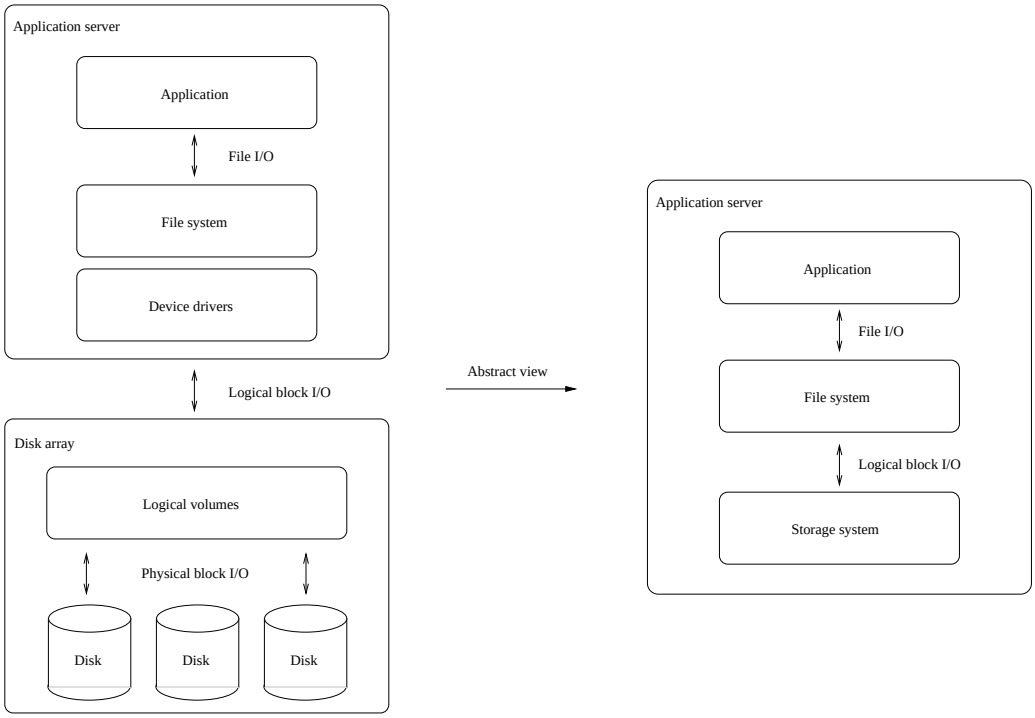

Figure 2.1: A canonical file and storage system (left half of figure) and an abstracted view (right half). An application issues file I/O to a file system, the file system translates file requests to logical block requests, the logical block requests are issued to a logical volume within a storage system, and the storage system maps logical blocks to physical blocks using some internal mapping (e.g., RAID).

.

application servers [27], like modularity, redundancy, and a "buy as you grow" model. Unfortunately, this trend has exacerbated the management problem, as managing multiple disk arrays is more difficult than managing a single array.

The already high cost of storage management, combined with the trend toward cluster-based storage, has sparked a flurry of activity around automated storage management, with an objective of automating many of the lower-level and time-consuming tasks. Automated storage management goes by many names, including system-managed storage [24], self-managed storage [19], autonomic storage [32], and self-* storage [22]. Names aside, there is a common goal: to offer the administrators some relief and reduce TCO. As examples, a storage system could configure its own RAID level, decide which workloads it will and will not host (based on its ability to deliver the proper QoS), adjust tunable parameters (e.g., cache eviction policies), and be proactive with respect to detecting and reporting security breaches.

2.3 Storage system design

Storage system design is an area of storage management with much potential for automation. It is a special case of the "generalized assignment problem" [13], where a collection of tasks (application workloads) are to be assigned to agents (logical volumes) that have varying capacity and performance

characteristics (e.g., different RAID levels). The generalized assignment problem is an NP-complete [23] optimization problem, meaning that it cannot be solved or verified in polynomial time. One of the earliest forms for computer storage is the classic "File Assignment Problem" [14, 18, 57, 43, 53, 9]. The problem is stated as such: given a collection of file servers, a collection of files and their access patterns, and a set of performance goals, what is the optimal assignment of files to file servers?

The focus of the problem has since shifted to storage systems, particularly disk arrays. However, the problem is still fundamentally one of bin-packing (i.e., "packing" workloads into disk arrays). Given a collection of application workloads and disk arrays, a solution to the generalized assignment problem involves the following:

1. Deciding which disk arrays to use, which may involve purchasing additional arrays.

2. Creating logical volumes within each disk array. Again, there are two decisions that must be made: the size of each volume and its type (linear, striping, or RAID).

3. Assigning I/O workloads to the logical volumes.

Though the above steps sound simple, there are an exponential number of possibilities, making the search space very large. Consider the number of ways one could configure 8 disk arrays for 8 workloads. Even with the simplifying assumption that each array can contain only a single volume configured as either RAID-1 (mirroring) or RAID-5 (striping with parity), there are still over 10 million possibilities (i.e., 2^8 ways of configuring the disk arrays by 8! ways of assigning each volume a workload.). Clearly, a system administrator does not have the luxury of physically trying each one. Instead, a prediction is usually made as to the best way to configure the arrays and assign them work. But, this is usually done in an *ad hoc* manner using industry rules-of-thumbs. In general, predicting the performance of a disk array for an arbitrary workload is difficult [4, 16, 37, 39]. It is even more difficult when multiple workloads are sharing the same array, as one must predict the effects of resource contention [11].

Given the dizzying number of configuration and assignment possibilities, the lack of information on the interaction between workloads and the disk arrays, and the risk of not meeting performance goals if the overall storage system is poorly designed, a common strategy is to simply over-provision [34] — that is, to buy more storage than one really needs and hope that it is good enough. This might be reasonable for smaller installations, but it is not practical for larger data centers, especially in light of the 50-100% yearly growth in storage capacity. Further, over-provisioning provides no guarantees that performance goals will be met.

2.4 An automated storage system

A better way to attack storage system design is to automatically determine the I/O characteristics of workloads and model the capabilities of the disk arrays relative to these characteristics. Then, one can make an informed decision as to the best assignment of workloads to storage. Of course, there is still the exponential number of possibilities, but optimization software [17] can be used to explore the search space. This involves using performance models to predict the performance of the disk arrays for various

workload assignments. Greedy heuristics are often used to guide the search (e.g., iteratively finding the logical volume that is the "best fit" for each workload).

In effect, one is mapping the characteristics of workloads to the capabilities of storage [47]. This was the general approach taken by earlier researchers with the File Assignment Problem and is also the approach used by storage systems researchers today. HP Labs, in particular, has produced multiple generations of automated storage system designers based on this concept, including Forum [10], Minerva [3], and Ergastulum [6]/Hippodrome [5]. Further, using the output from these design tools (i.e., a mapping of workloads to disk arrays), one can also automatically design the storage area network that will be used to connect the application servers to the disk arrays, so as to find a minimum-cost set of links to route the flows [30].

An automated storage system should also handle the transparent migration of data across (or within) disk arrays. There are numerous reasons for data migration, including disk array upgrades and maintenance, array failures, and workload change. Also, the first time a workload is characterized, it may not be on the "best" disk array. Therefore, after an optimization engine determines the proper assignment, the application data must be moved to that array, preferably without any application downtime or human intervention.

A variety of research has focused on automating the migration process. There are techniques for transparently migrating data between RAID levels [56] (e.g., RAID-1 vs. RAID-5) or different data distributions [1] (e.g., erasure coding vs. replication), techniques to minimize the impact to foreground traffic during a data migration [38], and techniques which involve cloning data sets [33]. There has also been research to minimize the number of migrations. In particular, if one can predict the I/O characteristics of workload, it can be assigned to the right logical volume from the start, thereby avoiding a potentially costly migration. One way to do this is to automatically classify files [20, 40] (e.g., read-only, write-mostly) when they are first created. This can be done using information such as the creation time, the filename, or the group and user ID (e.g., files created at 3 a.m. by "root" may be write-once-read-rarely backup files.). Once classified, a file's data can be allocated from a volume optimized for its predicted access pattern.

Figure 2.2 illustrates an automated storage system. The storage system is a cluster-based storage system composed of disk arrays. Logical volumes implement the "optimal mapping" of workloads to disk arrays, as per the instruction of the optimization engine. The workloads are applications such as file systems, databases, or any other application directly accessing the storage. The automation system (e.g., optimization engine, performance models, and data migration engine) may reside on a single management server or be distributed across servers or disk arrays. Irrespective of its system architecture, the goal of such an automation system is to provide application servers with logical volumes for storing and retrieving data, and this system is expected to evolve over time with the data center. For example, when a new disk array is purchased, performance models must be trained (offline), after which the disk array can be introduced into the storage cluster and automatically provisioned, thereby becoming part of a logical volume. Further, as new workloads are introduced, performance predictions must be made. When the predictions match the observed performance (i.e., a successful data migration), no additional work is necessary. Inaccurate predictions, however, will require that a model be automatically retrained, using the failed predictions as additional training data.

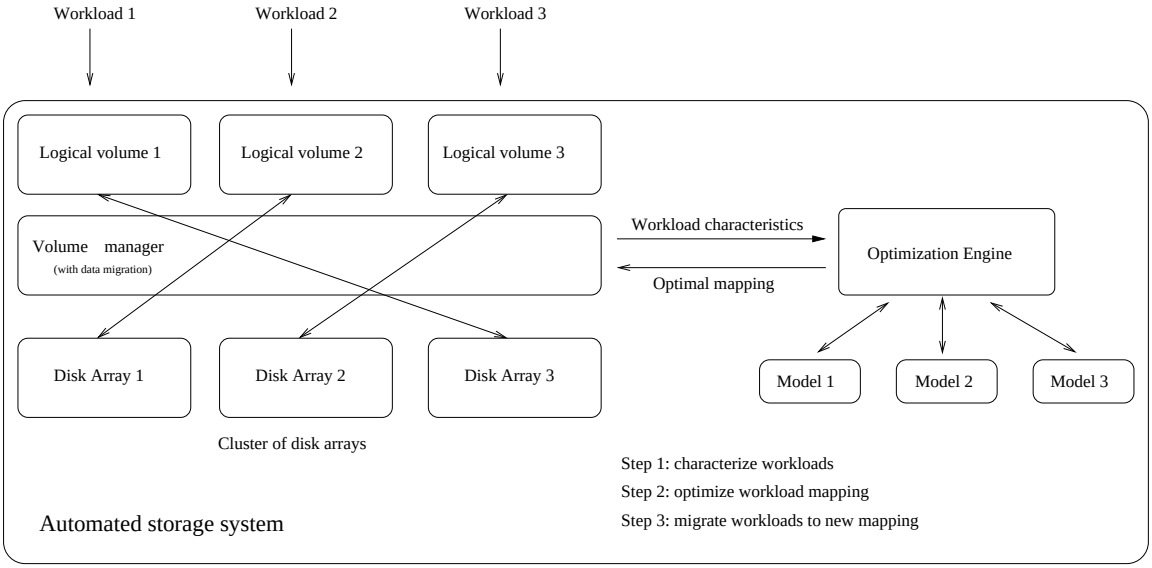

Figure 2.2: A workload is characterized the first time it is run, on the disk array to which it was originally assigned. Using these characteristics, an optimization solver can query each of the performance models and determine the optimal mapping. The workload can then be migrated to the assigned array. If the characteristics of the workload change, the process can be repeated.

Though not shown in Figure 2.2, there are a variety of other performance-related techniques that can be automated within a storage system, particularly as they relate to quality of service [8, 54, 55]. As examples, there could be different bandwidth allocation strategies in the storage system [46, 48, 49] (e.g., best-effort vs. soft real-time), performance differentiation among competing workloads [31], performance isolation among competing workloads [51, 52], and economic-based models where applications bid for storage resources [15].

2.4.1 Challenges

One of the biggest challenges with automated storage system design is characterizing and predicting the performance of workloads, and this challenge has been significant enough to keep an automated system, like that depicted in Figure 2.2, from being fully realized in practice. In short, performance models require concise workload characteristics, but it is difficult to make a workload characteristic concise without losing performance-critical information (a Catch-22).

More precisely, performance models require workload characteristics that can be mathematically manipulated, and these usually take the form of statistics, such as a workload's average request size or its read/write ratio. However, statistics can hide important information and misrepresent the real behavior of a workload. Conversely, one could use a non-statistical description of a workload (e.g., a complete I/O trace), but it would be difficult to use this information in a performance model.

It is in the process of "compressing" an I/O trace into workload characteristics that information is lost. That is, expressiveness and conciseness are at odds with one another, resulting in inaccurate performance predictions. This trade-off will be discussed further in Chapters 4 and 5.

2.5 Summary and concluding remarks

Determining how to best assign workloads to disk arrays is a task ideally suited to automation. In summary, one can characterize I/O workloads, model the performance of disk arrays, and use optimization software to assign workloads to the proper arrays. For each possibility, the performance models can be used to predict performance, and the optimization software can proceed to find an assignment with the best overall performance or perhaps the best price-performance.

Though simply stated, this bin-packing problem has challenged storage system researchers for over 30 years, beginning with the File Assignment Problem. Perhaps the biggest challenge is that of workload characterization. To be used by the performance models, one must be able to describe a workload in a way that is both expressive and concise.

In many respects, the most expressive and concise description of a workload is its performance on a given storage system. Indeed, the performance of one storage system may be correlated with another, and performance is exactly what we want to predict. For example, the average I/O latency of a workload can indicate the effectiveness of a cache and give some indication of the spatio-temporal access pattern. In addition, one could describe a workload by its resource utilization (e.g., the CPU, memory, and network resources that are consumed in the storage system).

Unfortunately, performance and resource utilization metrics are specific to a given storage system and are usually discarded when building models and making predictions. This is because, conventionally, only one model is built per storage system. As such, the workload characteristics used by a model must be independent from the storage system on which they were obtained. This, of course, precludes the use of performance and resource utilization.

A primary motivation behind relative fitness modeling is finding a way to retain this information, thereby reducing the dependence on workload characterization. As will be described in Chapter 6, this is accomplished by modeling storage systems relative to one another. In effect, one can regard performance and utilization as very concise, performance-critical workload characteristics.

Bibliography

[1] Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Ursa Minor: versatile cluster-based storage. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, December 2005. The USENIX Association.

[2] N. Allen. Don't waste your storage dollars: what you need to know, March 2001. Gartner Group.

[3] Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. MINERVA: an automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems (TOCS)*, 9(4):483–518, November 2001.

[4] Guillermo A. Alvarez, Walter A. Burkhard, Larry J. Stockmeyer, and Flaviu Cristian. Declustered disk array architectures with optimal and near-optimal parallelism. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA 1998)*, Barcelona, Spain, June 1998.

[5] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: running circles around storage administration. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST 02)*, Monterey, CA, January 2002. The USENIX Association.

[6] Eric Anderson, Mahesh Kallahalla, Susan Spence, Ram Swaminathan, and Qian Wang. Quickly finding near-optimal storage system designs. *ACM Transactions on Computer Systems (TOCS)*, 23(4):337–374, November 2005.

[7] Network Appliance. PostMark: A New File System Benchmark. http://www.netapp.com.

[8] Cristina Aurrecoechea, Andrew T. Campbell, and Linda Hauw. A survey of QoS architectures. *Multimedia Systems*, 6(3):138–151, May 1998.

[9] Baruch Awerbuch, Yair Bartal, and Amos Fiat. Competitive Distributed File Allocation. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC 93)*, San Diego, CA, May 1993. ACM Press.

[10] E. Borowsky, R. Golding, A. Merchant, L. Schreier, E. Shriver, M. Spasojevic, , and J. Wilkes. Using attribute-managed storage to achieve QoS. In *Proceedings of the 5th International Workshop on Quality of Service (IWQoS 1997)*, New York, NY, May 1997.

[11] Elizabeth Borowsky, Richard Golding, Patricia Jacobson, Arif Merchant, Louis Schreier, Mirjana Spasojevica, and John Wilkes. Capacity planning with phased workloads. In *Proceedings of the First Workshop on Software and Performance (WOSP98)*, Santa Fe, New Mexico, October 1998. ACM Press.

[12] Aaron Brown, David Oppenheimer, Kimberly Keeton, Randi Thomas, John Kubiatowick, and David A. Patterson. ISTORE: Introspective Storage for Data-Intensive Network Services. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, Arizona, March 1999. The USENIX Association.

[13] D.G. Cattrysse and L. N. Van Wassenhove. A Survey of Algorithms for the Generalized Assignment Problem. *European Journal of Operational Research*, 60:260–272, June 1992.

[14] K. M. Chandy and J. E. Hewes. File allocation in distributed systems. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1976)*, New York, NY, March 1976. ACM Press.

[15] Jeffrey Chase, Darrell Anderson, Prachi Thakar, Amin Vahdat, and Ronald Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP 01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[16] Shenze Chen and Don Towsley. A performance evaluation of RAID architectures. *IEEE Transactions on Computers*, 45(10):1116–1130, October 1996.

[17] Joseph Czyzyk, Michael P. Mesnier, and Jorge Moré. The NEOS Server. *IEEE Computational Science and Engineering*, 5(3):68–75, July-September 1998. http://neos.mcs.anl.gov.

[18] Lawrence W. Dowdy and Derrell V. Foster. Comparative Models of the File Assignment Problem. *ACM Computing Surveys*, 14(2):287–313, June 1982.

[19] Elizabeth Borowsky and Richard Golding and Arif Merchant and Elizabeth Shriver and Mirjana Spasojevic and John Wilkes. Eliminating storage headaches through self-management. In *Proceedings of the 2nd Symposium on OS Design and Implementation (OSDI'96)*, Seattle, WA, October 1996. The USENIX Association.

[20] Daniel Ellard, Michael Mesnier, Eno Thereska, Gregory R. Ganger, and Margo Seltzer. Attribute-based prediction of file properties. Technical Report TR-14-03, Harvard University, December 2003.

[21] Svend Frølund, Arif Merchant, Yasushi Saito, Susan Spence, and Alistair Veitch. FAB: enterprise storage systems on a shoestring. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Lihue, HI, May 2003. The USENIX Association.

[22] Gregory R. Ganger, John D. Strunk, and Andrew J. Klosterman. Self-* Storage: Brick-based Storage with Automated Administration. Technical Report CMU-CS-03-178, Carnegie Mellon University, August 2003.

[23] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, NY, 1979.

[24] J. P. Gelb. System-managed storage. *IBM Systems Journal*, 28(1):77–103, 1989.

[25] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A Cost-effective, High-Bandwidth Storage Architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, October 1998. ACM Press.

[26] Jim Gray. A Conversation with Jim Gray. *ACM Queue*, 1, 2003.

[27] Jim Gray and Daniel P. Siewiorek. High-availability computer systems. *IEEE Computer*, 24(9):39–48, September 1991.

[28] Gartner Group. Total Cost of Storage Ownership: A User-oriented Approach, February 2000.

[29] PC Guide. Redundant Arrays of Inexpensive Disks (RAID). www.pcguide.com/ref/hdd/perf/raid/.

[30] Julie Ward and Michael O'Sullivan, and Troy Shahoumian and John Wilkes. Appia: automatic storage area network fabric design. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST 02)*, Monterey, CA, January 2002. The USENIX Association.

[31] Magnus Karlsson, Christos Karamanolis, and Xiaoyun Zhu. Triage: performance differentiation for storage systems using adaptive control. *ACM Transactions on Storage*, 1(4):457–480, November 2006.

[32] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer Magazine*, 36(1):41–50, January 2003.

[33] Samir Khuller, Yoo-Ah Kim, and Yung-Chun (Justin) Wan. Algorithms for data migration with cloning. In *Proceedings of the 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, San Diego, CA, June 2003. ACM Press.

[34] Steve Kleiman. Trends in Managing Data at the Petabyte Scale (Invited Talk). In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 07)*, San Jose, CA, February 2007. http://www.usenix.org/events/fast07/tech/slides/kleiman.pdf.

[35] E. Lamb. Hardware spending matters. *Red Herring*, pages 32–33, June 2001.

[36] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Cambridge, MA, October 1996. ACM Press.

[37] E.K. Lee and R.H. Katz. Performance consequences of parity placement in disk arrays. In *Proceedingsof the 4th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1991)*, Santa Clara, CA, April 1991.

[38] Chenyang Lu, Guillermo A. Alvarez, and John Wilkes. Aqueduct: online data migration with performance guarantees. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST 02)*, Monterey, CA, January 2002. The USENIX Association.

[39] Jay Menon. Special issue on disk arrays - introduction. *Distributed and Parallel Databases*, 2(3), July 1994.

[40] Michael Mesnier, Eno Thereska, Greg Ganger, Daniel Ellard, and Margo Seltzer. File classification in self-* storage systems. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC-04)*, New York, NY, May 2004. IEEE Computer Society.

[41] Panasas. PanFS Parallel File System. http://www.panasas.com.

[42] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the International Conference on Management of Data (ACM SIGMOD 1988)*, Chicago, IL, June 1988. ACM Press.

[43] Krisha R. Pattipati and Joel L. Wolf. A File Assignment Problem Model for Extended Local Area Network Environments. In *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS 1990)*, Paris, France, May 1990. IEEE Computer Society.

[44] Nemertes Research. The New Data Center, 2006.

[45] Drew Robb. Lowering Storage TCO at Cisco. *Enterprise IT Planet*, June 21 2004.

[46] Prashant Shenoy and Harrick M. Vin. Cello: A Disk Scheduling Framework for Next Generation Operating Systems. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1998)*, Madison, WI, June 1998. ACM Press.

[47] Elizabeth Shriver. A formalization of the attribute mapping problem. Technical Report HPL-SSP-95-10 Rev D, Hewlett-Packard Laboratories, July 1996.

[48] Vijay Sundaram and Prashant Shenoy. Bandwidth Allocation in a Self-Managing Multimedia File Server. In *Proceedings of the Ninth ACM International Conference on Multimedia (ACM Multimedia 2001)*, Ottawa, Ontario, Canada, October 2001.

[49] Vijay Sundaram and Prashant Shenoy. A Practical Learning-based Approach for Dynamic Storage Bandwidth Allocation. In *Proceedings of the Eleventh International Workshop on Quality of Service (IWQoS 2003)*, Berkeley, CA, June 2003. Springer.

[50] Cluster File Systems. Lustre File System. http://www.clusterfs.com.

[51] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance Insulation for Shared Storage Servers. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 07)*, San Jose, CA, February 2007. The USENIX Association.

[52] Yin Wang and Arif Merchant. Proportional share scheduling for distributed storage systems. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 07)*, San Jose, CA, February 2007. The USENIX Association.

[53] Gerhard Weikum, Peter Zabback, and Peter Scheuermann. Dynamic File Allocation in Disk Arrays. In *Proceedings of the International Conference on Management of Data (ACM SIGMOD 1991)*, Denver, CO, May 1991. ACM Press.

[54] Ravi Wijayaratne and A. L. Narasimha Reddy. Providing QoS guarantees for disk I/O. *Multimedia Systems*, 8(1):57–68, February 2000.

[55] John Wilkes. Traveling to Rome: QoS specifications for automated storage system management. In *Proceedings of the 9th International Workshop on Quality of Service (IWQoS 2001)*, Karlsruhe, Germany, June 2001.

[56] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems (TOCS)*, 14(1):108–136, February 1996.

[57] Joel Wolf. The Placement Optimization Program: A Practical Solution to the Disk File Assignment Problem. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1989)*, Berkeley, CA, June 1989. ACM Press.

[58] Zheng Zhang, Shiding Lin, Qiao Lian, and Chao Jin. RepStore: A Self-Managing and Self-Tuning Storage Backend with Smart Bricks. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC-04)*, New York, NY, May 2004. IEEE.

Chapter 3

Measuring storage performance

"*If you can not measure it, you can not improve it.*" Lord Kelvin.

A variety of metrics are used to evaluate storage systems, including performance, capacity, availability, reliability, and cost. Although performance is not the only measure of success, any new storage system will certainly be judged on its speed. Commonly used performance metrics include I/O bandwidth (data transfer rate), throughput (I/O request completion rate), and latency (I/O request execution time).

This chapter introduces performance metrics commonly used in evaluating storage systems. Section 3.1 describes the performance metrics in more detail and points where they can be measured, including the file level and block level. Section 3.2 addresses some of the measurement pitfalls encountered in this research and provides tips for avoiding them. In effect, Section 3.2 establishes the rules by which performance will be measured in the Chapter 7 evaluation, where relative fitness models are trained to predict the performance of various workloads. Section 3.3 introduces various benchmarks used throughout the storage industry and how they are used to generate I/O workloads from which performance measurements can be taken.

3.1 Performance metrics

Three performance metrics are commonly used to quantify storage system performance. These are the I/O bandwidth, throughput, and latency of a workload. Although terminology varies somewhat throughout the industry, the working definitions are primarily the same. In particular, the Storage Networking Industry Association (SNIA) [4] defines bandwidth as the "data transfer capacity," or the maximum rate in which data can be transmitted. Throughput is the number of I/O operation requests that can be satisfied per unit time. Latency is the request execution time. Of course, bandwidth and throughput are both averages over some period of time. Latency is often reported as an average, but one could also report any percentile (e.g., the 90th percentile of request latency).

In the context of a file system, an "I/O" is a file system operation (e.g., `open()`, `close()`, `read()`, or `write()`). Bandwidth measurements apply to the read and write data transfer rates for files, throughput measurements are typically broken out into individual operations (e.g. the maximum file creation rate), and latency measurements can apply to any of the file operations (e.g., the average latency for a file

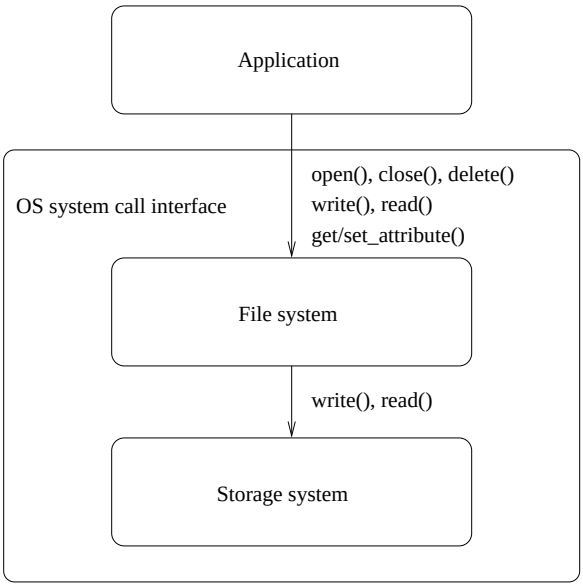

Figure 3.1: Applications interface with the file system through OS system calls. The file system interfaces with the storage system using `write()` and `read()` commands that specify the blocks to be written/read.

creation). For storage systems, however, an I/O is simply a read or a write operation. Figure 3.1 illustrates the relationship between file and block I/O within an operating system.

Performance can be measured at the interface between the application and the file system (the file level), or the file system and the storage system (the block level). At the file level, performance counters can be used to measure the end-to-end performance of the file system. This is accomplished by instrumenting the application or the file system, or by tracing the I/O between an application and the file system. In contrast, counters at the block level measure the end-to-end performance of the storage system, which will also include I/O related to file system metadata. For the purpose of storage system modeling and performance prediction, block-level performance measurements are commonly used. The block level represents the lowest level in the stack — below all applications, file systems, and OS caches. As such, it represents the workload as observed by a storage system.

The following sections discuss bandwidth, throughput, and latency in more detail. Each of these metrics relies on performance counters in the storage system (e.g., total bytes transferred and total I/Os completed) to calculate the amount of work that has been completed over a specified period of time, referred to as a *measurement window* in this dissertation. Counter values are recorded at the beginning and end of the measurement window. To calculate the total amount of work completed, one simply subtracts the first set of counters from the second set. Tips for establishing a proper measurement window are discussed in Section 3.2.

The metrics and their required counters are now discussed. It is assumed that the two sets of counters have been read and that their differences have been calculated. More detail on using such counters is presented in Chapter 7.

3.1.1 Bandwidth

Bandwidth is the data transfer rate. It represents the average number of bytes transferred (read or written) over a unit of time. The common unit today is megabytes per second (MB/sec). This measurement is obtained by dividing the total bytes transferred (megabytes) over the length of the measurement window (seconds). Instantaneous bandwidth for a single I/O operation could also be reported, but this is not commonly done in practice.

Only one counter is required to calculate bandwidth: a count of the total bytes written and read. Below is the very simple equation for bandwidth:

$$Bandwidth = \frac{Total\ number\ of\ bytes\ transferred}{Length\ of\ measurement\ window} \tag{3.1}$$

3.1.2 Throughput

Throughput is the I/O request completion rate. It represents the average number of I/O operations completed per unit of time. The most common unit is the number of I/O operations completed per second, often abbreviated as IOPS. Only one counter is required to calculate throughput: a count of the operations completed. Throughput is calculated by dividing the number of completed operations by the length of the measurement window:

$$Throughput = \frac{Total\ number\ of\ I/Os\ completed}{Length\ of\ measurement\ window} \tag{3.2}$$

A common mistake in calculating throughput is to simply take the inverse of the request execution time (latency). For example, if the average request latency is 10 ms, then one might (incorrectly) conclude that the throughput is 1 I/O every 10 ms, or 100 IOPS. However, this is only correct when the multi-programming level is one (i.e., no command queuing), there is no "wait time" for an I/O request, and there is no "think time" (computation time) between I/Os. Note, in the context of storage systems, the multi-programming level is a measure of I/O concurrency. It refers to the number of I/O operations allowed to enter into the storage system, not necessarily the number of "service stations" or storage devices that are prepared to service the I/O. When the multi-programming level is greater than one, there may be some amount of wait time for each I/O in the storage system queues. Similarly, if the think time is non-zero, there will be some amount of computation (or idle time) before the next I/O is issued by the application.

3.1.3 Latency

Latency is the average I/O request time. It represents the average duration of an I/O operation (i.e., the time from when an I/O operation is issued to the storage system to when it completes). In queuing theory, the request time is the sum of the I/O wait time (i.e., waiting to use components of the storage system) and the service time (i.e., executing in the storage system). The latency calculation is more involved than that of bandwidth or throughput. One approach is to sum the request times of all completed I/O operations over a measurement window and divide the sum by the total number of completed I/O operations:

$$Latency = \frac{\sum RequestTime_i}{Total\ number\ of\ I/Os\ completed} \tag{3.3}$$

Similarly to throughput, a common mistake in calculating latency is to simply take the inverse of the throughput measurement, which produces the average I/O request inter-arrival (or inter-completion) time. For example, if the throughput of a system is 100 I/O operations per second, the average inter-arrival time of each I/O is 1/100 of a second or one I/O every 10 milliseconds (ms). Again, this is only equivalent to the I/O latency when the multi-programming level is one and the think time is zero.

3.2 Measurement pitfalls

At least two pitfalls are encountered when measuring the performance of storage systems: variance in the measurements and deceptive averages. Variance can make performance prediction difficult. Indeed, variance is the nemesis of predictability — if results are not repeatable, they are definitely not predictable. Further, averages can mask the true I/O behavior of an application. For example, if an application runs for 24 hours and issues all of its I/O in the last hour, it would be misleading to measure and report I/O performance over the 24 hour period, as the performance would be deceptively low.

In general, such pitfalls are due to variability in the I/O workload. I/O can be bursty (e.g., write bursts from the file system cache), cyclic (e.g., cache flushes every 30 seconds), and multi-phased (e.g., a compute phase, followed by write phase, and then a read phase). When measuring the performance of a workload on a particular storage system, one must be aware of each of these possibilities and measure performance over the desired period. This is done by specifying the proper measurement window.

The measurement window is specified by two parameters. The first parameter is the *warming* time, or the amount of time before the first set of performance counters is read. In this time, the storage system is allowed to "warm" and reach a steady state of performance. The second parameter is the *testing* time, or the amount of time the application is allowed to run before the second set of counters is read.

In general, if the goal of the performance measurement is to capture the steady-state performance of a system, the warming time must be chosen large enough that all performance-affecting components (e.g., caches) have been sufficiently warmed, and the testing time chosen long enough to establish a stable average, and no longer, else one might be measuring across multiple I/O phases of an application. For example, the Postmark Benchmark [3] is composed of two phases: file pool creation and file transactions (e.g., read, append, delete). If one were to measure performance over the entire run of Postmark, the measurement would reflect average performance over the two phases — a potentially meaningless measurement. Instead, it is more useful to measure and report the two phases separately.

Of course, completely avoiding these pitfalls requires some awareness of the application's I/O workload (i.e., its workload characteristics). Workload characterization is introduced in detail in the next chapter. However, one quick way of learning about the I/O demands of an application is to visualize its performance over time, using a graphical performance monitor. Such a tool can show I/O bursts or cyclic activity and is useful in establishing a good measurement window. In particular, a graphical performance monitor will show if the performance of an application workload "levels off" and reaches a steady-state and if there are different I/O phases. Often, a phase change will correspond to sudden (visual) changes in the I/O performance. For example, Figure 3.2 shows the bandwidth of Postmark over time. One can see two distinct I/O phases, corresponding to Postmark's file creation phase and its transactions phase.

Assuming a good measurement window as been chosen (i.e., the application has warmed for sufficiently

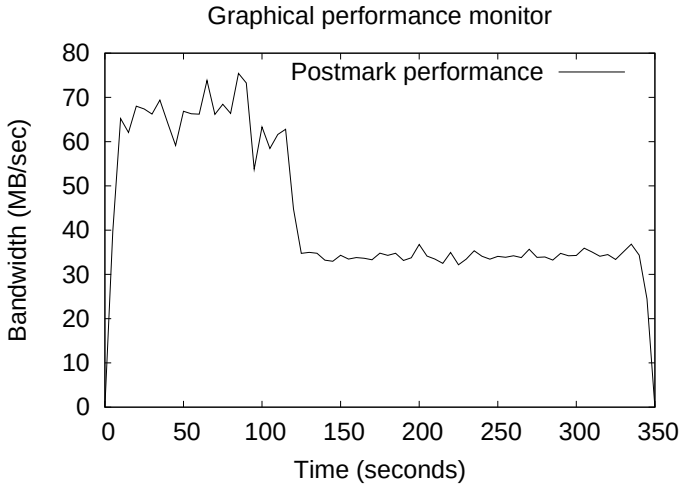

Figure 3.2: A screen shot from a graphical performance monitor [7]. The graph plots the bandwidth of Postmark over time, illustrating two distinct I/O phases (file creation and file transactions).

long, and a single I/O phase is being measured), one can still run into difficulties with averages. This is particularly the case with I/O latency. Because latency is an average over all I/O operations, a few outliers (with extremely high latency) can increase the average considerably. In cases such as these, it can be better to report the median (or a percentile), rather than the mean. The distribution of I/O latency can be consulted if one is concerned whether outliers are affecting the average.

In general, one should pick meaningful measures for the statistical dispersion, or the "spread," of the data. For performance analysis and prediction, it is always good to take multiple measurements (when possible) and compute their variance, because variance will limit the predication accuracy of the models. For example, if the measured performance of an I/O workload varies by 10%, one cannot reasonably expect a performance model to predict with an accuracy that is better than 10%. For storage systems, it is useful to have a least two measures of variance: an absolute measure (e.g., the bandwidth measurements varied by 3 MB/sec) and a relative measure (e.g., the bandwidth measurements varied by 30%). The mean absolute deviation (an absolute measure) and the coefficient of variation (a relative measure) are useful statistics for making such measurements. In addition, the range (i.e., the difference between the minimum and maximum values) gives the maximum spread of a set of measurements. Each of these statistics will be used in the evaluation of the relative fitness models in Chapter 7.

3.3 Available benchmarks

Many benchmarks have been developed throughput the years for the purposes of file and storage system evaluation. In addition to reporting application performance metrics (e.g., compilation time or the database transaction rate), benchmarks are often used to generate I/O workloads from which bandwidth, throughput, and latency measurements can be taken. There are micro-benchmarks that test specific functionality (e.g., file creation rate, or sequential I/O performance) and macro-benchmarks composed

of either real or simulated application workloads (e.g., kernel compilation or database transactions). Benchmarks can further be described as being either file-level benchmarks (testing the file and storage system) or block-level (testing only the storage system).

Popular macro-benchmarks for file systems include Andrew [10] (source code manipulation and compilation), Postmark [3] (news, e-mail, and WWW), TPC-C [8] (online transaction processing), and TPC-H [9] (decision support). Both TPC-C and TPC-H are database benchmarks created by the Transaction Processing Performance Council (TPC). They test random I/O performance (TPC-C) and sequential I/O performance (TPC-H). In addition, model-based benchmarking frameworks are beginning to emerge (e.g., Filebench [13]), where one can model workloads like that of an Internet Service Provider.

Popular micro-benchmarks for file systems include Bonnie [5] and IOzone [14]. Both report the performance of various I/O operations to a single file. There are also self-scaling [6] micro-benchmarks specifically designed for NFS file servers, including SPECsfs [15] and Fstress [1]. A self-scaling benchmark will adjust its workload parameters in response to the performance of the system being measured.

Many of the file system benchmarks offer a variety of application-specific performance metrics that can be used to quantify the storage system performance. For example, the Andrew benchmark reports the amount of time necessary to copy, read, and compile a source code repository, Postmark reports the average file transaction rate (creates, deletes, reads, and writes), TPC-C reports the number of transactions per minute (tpmC), and TPC-H reports a composite number that reflects the number of database queries per hour, the database size, single-client query performance, and multi-client query performance. However, when describing (or predicting) the performance of a storage system, the lower-level metrics (bandwidth, throughput, and latency) are usually preferred.

For storage systems, the Storage Performance Council (SPC) has developed macro-benchmarks (SPC-1 and SPC-2). These benchmarks are similar to TPC-C and TPC-H, respectively, in that one measures random I/O throughput and the other sequential. However, the SPC benchmarks are intended only for storage system benchmarking. They are not allowed, as per the SPC guidelines, to utilize the services of a file system (e.g., caching and read-ahead).

Buttress [2], Fitness [7], Iometer [11] and Xdd [12] are synthetic workload generators. When configured in a specific manner, they can be used as storage system micro-benchmarks. Commonly, such tools are used by varying input parameters such as the read/write ratio, the amount of spatial randomness, the read/write request sizes, and the level of I/O concurrency.

Traeger and Zadok provide an in-depth study of many of these benchmarks are how they are commonly used and reported in the literature [16]. New or revised benchmarks appear every year or so. As examples of the frequency, these are the release dates for the benchmarks just described: Andrew (1988), Bonnie (1989), TPC-C (1992), SPECsfs (1993), Postmark (1997), Iometer (1998), IOzone (1998), TPC-H (1999), SPC-1 (2001), Fstress (2002), Buttress (2004), Xdd (2005), SPC-2 (2005), Filebench (2005), and Fitness (2006).

Postmark, TPC-C, and Fitness will be discussed in greater detail in Chapter 7, where relative fitness models are trained to predict their performance. As will be explained, one can create multiple workload samples from each of these benchmarks, simply by varying their input parameters.

3.4 Summary and concluding remarks

Obtaining good performance measurements is a critical first step in effectively modeling and predicting the performance of storage systems. This chapter described three block-level metrics that are commonly used (bandwidth, throughput, and latency) and provided some tips for establishing a proper measurement window. In particular, one should allow an application to reach a steady-state of performance in a storage system, measure the performance of a single phase of the application, and measure over multiple runs to check for variance that might limit the accuracy of performance models.

Although a variety of benchmarks were discussed, which can be used to create workloads from which storage system performance can be measured, one could argue that the end-to-end performance of a real application is the only measurement that really matters when it comes to performance prediction. One could also argue that real applications often fail to reach a steady state of I/O performance in a storage system, but are characterize more by intermittent bursts of I/O — thereby making a "good" performance measurement impractical. The author agrees on both counts and assumes that, for such applications, measuring the average I/O performance (or perhaps a percentile on I/O latency) is useful in predicting which storage system is best for a workload, even if the measurement is not cleanly taken over a single, steady-state I/O phase as described.

Bibliography

[1] Darrell Anderson and Jeff Chase. Fstress: A Flexible Network File Service Benchmark. Technical report, Department of Computer Science, Duke University, 2002.

[2] Eric Anderson, Mahesh Kallahalla, Mustafa Uysal, and Ram Swaminathan. Buttress: A toolkit for flexible and high fidelity I/O benchmarking. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST 04)*, San Francisco, CA, March 2004. The USENIX Association.

[3] Network Appliance. PostMark: A New File System Benchmark. http://www.netapp.com.

[4] Storage Networking Industry Association. A Dictionary of Storage Networking Terminology. http://www.snia.org/education/dictionary.

[5] Tim Bray. The Bonnie Benchmark. http://www.textuality.com.

[6] Peter M. Chen and David A. Patterson. A New Approach to I/O Performance Evaluation - Self-Scaling I/O Benchmarks. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2007)*, Santa Clara, CA, May 10–14 1993. ACM Press.

[7] Intel Corporation. Open Storage Toolkit. http://www.sourceforge.net/projects/intel-iscsi.

[8] Transaction Processing Performance Council. TPC Benchmark C. http://www.tpc.org/tpcc.

[9] Transaction Processing Performance Council. TPC Benchmark H. http://www.tpc.org/tpch.

[10] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, Mahadev Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, February 1988.

[11] Intel Corporation. Iometer. http://www.iometer.org.

[12] I/O Performance Inc. Xdd. http://www.ioperformance.com.

[13] Richard McDougall, Joshua Crase, and Shawn Debnath. FileBench, 2005. http://sourceforge.net/projects/filebench.

[14] William Norcott and Don Capps. IOzone Filesystem Benchmark. http://www.iozone.org.

[15] Standard Performance Evaluation Corporation. Spec sfs97 v3.0. http://www.storageperformance.org.

[16] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P. Wright. A Nine Year Study of File System and Storage Benchmarking. Technical Report FSL-07-01, Computer Science Department, Stony Brook University, May 2007.

Chapter 4

Characterizing storage workloads

> "*Not everything that counts can be counted, and not everything that can be counted counts.*"
> Albert Einstein.

Workload characterization is the process of describing the nature of the work performed (or to be performed) by a system. In the context of storage systems, workload characteristics quantify an application's I/O demand, including information on operation mixes, temporal and spatial access patterns, and concurrency. In general, one rarely discusses performance without also mentioning the type of workload.

Workload characterization is a well-researched area, both for file systems [3, 6, 14, 15, 16, 18, 19] and storage systems [9, 10, 11, 12, 17, 22, 24]. The central goal is to describe an application's I/O in as *absolute* a way as possible, such that the characteristics will not be affected by the storage system on which the application is running. For example, a read-only workload will always be read-only, and the storage system on which it runs will do nothing to change this.

In addition to explaining and reasoning about the performance of a storage system, workload characteristics are also used in modeling and predicting performance. Specifically, to predict the performance of a workload on a given storage system, one inputs the characteristics of the workload into a performance model of the storage system. Of course, the accuracy of the model will be, in no small part, determined by the quality of the workload characteristics. As described in Chapter 2, such models can be used in automated storage system design (e.g., setting the appropriate RAID level of a storage array [25]). Workload characteristics are also used for synthetic workload generation [2, 5]. It is often easier to work with a workload generator than the actual application. However, like models, a synthetic workload generator is only as good as the workload characteristics used to describe the I/O [7]. Performance models and workload generators will be discussed in more detail in Chapter 5.

In effect, workload characterization is a form of compression. Given a trace of an application's I/O, including the time of each request, the operation type (e.g., read vs. write), and the arguments (e.g., disk ID, block offset, length), a workload characterization tool (e.g., Rubicon [20]) will try to "compress" the trace into meaningful characteristics. It is much easier to discuss a workload using characteristics such as its read/write ratio, average request size, and multi-programming level, as opposed to referring someone to an I/O trace.

4.1 Workload characteristics

Decades of storage systems research have identified numerous workload characteristics that can be used to describe I/O. Common among them are measures of the read/write ratio, I/O request sizes, spatial locality (e.g., average number of blocks skipped, or "jumped," between successive I/O requests), temporal locality (e.g., I/O arrival rate), and concurrency (e.g., the number of concurrent streams and the multi-programming level of each stream). Other, more descriptive, characteristics include the temperature of data [4] (the ratio of I/O throughput to capacity), the inter-arrival time of I/O bursts [1], the phasing (overlap) of different I/O streams [1], measures of temporal burstiness [9, 24] (I/Os, especially writes, often come in bursts), measures of spatial burstiness [10, 22] (a burst might only affect certain blocks on disk), and spatio-temporal correlations [22] (certain blocks are accessed at certain times).

Note that the same performance measurement points discussed in Chapter 3 (file or block level) apply to workload characteristics. When characterizing I/O at the block level (e.g., in a SCSI device driver), one is able to obtain the aggregate I/O workload of all applications using a storage system. In contrast, at the application or file level, one would need to somehow combine the characteristics of multiple applications. Doing so, even with a detailed knowledge of the operating system, is challenging. For example, one would need to predict how write requests are aggregated across I/O streams, if at all, and how the elevator algorithms schedule I/O requests. Further, characteristics at the application and file level contain no metadata, which can significantly influence I/O performance. Therefore, when modeling and predicting the performance of storage systems, the block level is the most convenient level to work at. The trade-off is the lack of stream detection. At the block-level, one cannot distinguish the I/O of different applications or processes. However, if the goal is to predict the overall performance of a storage system, this does not present a problem.

The sections below describe a variety of block-level workload characteristics in greater detail and how they are useful in modeling and predicting performance. Equations are provided for the workload characteristics used in the evaluation of the relative fitness models in Chapter 7.

4.1.1 Read/write ratio

Write and read requests are handled differently by most storage systems. Whereas write requests are delayed, aggregated, and scheduled within a write-back cache, read requests are serviced on-demand. Of course, read-ahead and caching prevent many reads from going to disk, and scheduling algorithms (e.g., elevator) can efficiently order the read requests that do go to disk. Still, the freedom to optimize read requests is limited by the fact that there usually are not as many of them outstanding.

Because of these performance differences, it is important to distinguish reads from writes, beginning with the read/write ratio. In addition, many of the other workload characteristics are maintained separately for write and read requests (e.g., the write request size is measured independently of the read request size). As will be shown in Chapter 7, when compared to the read characteristics, the write characteristics are not as critical in predicting storage system performance.

A write fraction can be expressed as follows:

$$Write\ request\ fraction = \frac{Number\ of\ write\ requests}{Number\ of\ write\ requests + Number\ of\ read\ requests} \tag{4.1}$$

One could also capture the write data fraction:

$$Write\ data\ fraction = \frac{Number\ of\ bytes\ written}{Number\ of\ bytes\ written + Number\ of\ bytes\ read} \tag{4.2}$$

The read request fraction can be obtained by subtracting the write request fraction from 1, and similarly for the read data fraction. However, for the purposes of performance modeling and prediction, it is not necessary to maintain these characteristics for both writes and reads. One can be derived from the other and maintaining both would be redundant.

4.1.2 I/O request sizes

The smallest amount of data that can be accessed in a block-based storage system is a block, which is typically 512 bytes. However, much more efficient accesses can be made with larger requests that amortize the cost of each I/O request over a larger transfer of data.

The average request size of a workload is an important characteristic when predicting performance and should be calculated separately for reads and writes. Unlike writes, read requests are not as likely to be coalesced (because they are not delayed). Therefore, the read performance can be more sensitive to the characteristics of the read requests than the write performance is to that of the write requests. For example, when data is being read, the only requests eligible for coalescing are those currently outstanding in the queues of the storage system. In the case of, say, random I/O, very little coalescing will occur for reads. However, for writes, all requests may first hit in a write-back cache (e.g., in the host OS and/or in the storage system). From there, small random writes can be delayed, possibly coalesced, and reordered. Indeed, with a large enough write-back cache, all small random writes could be converted to large sequential writes.

The average write request size is simply the total number of bytes written divided by the total number of write requests, and the average read request size is calculated in the same manner:

$$Write\ request\ size = \frac{Number\ of\ bytes\ written}{Number\ of\ write\ requests} \tag{4.3}$$

4.1.3 Spatial randomness

The most efficient spatial access pattern for a disk-based storage system is usually that of sequential I/O, where the maximum data transfer rate is determined by each disk's rotation speed, track density, and bus speed. Conversely, the worst performance usually occurs for randomly accessed data, where performance is limited by positioning (seek) times and rotational delays.

One way to measure the spatial access pattern is to calculate the average "jump" distance for each I/O, or the number of blocks that are skipped between successive I/Os. For a sequential stream, the average jump will be zero. For a uniformly random stream, the jump distance will, on average, be one

third of the block range [1]. Therefore, one can normalize by 1/3 of the block range to create a single "randomness" value. For example, if 100 blocks are accessed 100% randomly, the average jump distance will be 33 blocks per I/O. If an I/O stream only had an average jump distance of 16.5 blocks per I/O, then it would be 50% random (i.e., 16.5/33).

To calculate the average jump distance, one must keep track of the last block read or written in the storage system. For each I/O request, the block distance relative to the last I/O is added to a counter. Then, to calculate the average jump distance over a collection of I/O requests, one simply divides the counter by the number of requests observed over a certain measurement period:

$$Jump = \frac{Jump\ counter}{Number\ of\ I/O\ requests} \tag{4.4}$$

One can also calculate jump distances separately for reads and writes, where the write jumps are calculated relative to the last write and the read jumps relative to the last read. This can be more telling than a single randomness measure, as the read randomness can be unaffected by the write randomness (due to write-back caching). As will be shown in Chapter 6, the read jump distance is a much better predictor of performance than the write jump distance. Again, the write requests have more potential for optimization in a storage system, making performance less sensitive to their spatial randomness.

4.1.4 I/O concurrency

There are two types of I/O concurrency in a storage system: multiple outstanding commands for a given I/O stream (e.g., an application reading a file, multiple requests at a time) and multiple I/O streams (e.g., multiple applications reading multiple files). Both types affect the performance of a storage system. However, one is usually for the better and the other for the worse.

For the better: command queuing

In the case of a single I/O stream, increasing the multi-programming level via command queuing can increase the efficiency of the storage system because of more efficient I/O scheduling. It can also substantially improve the performance of sequential I/O. This second point deserves more explanation, as it is often overlooked.

Consider the case where an application is writing a file stored on a single disk platter, one block at a time, with a multi-programming level of one and no write-back caching. As such, every I/O will incur the full rotational latency of the disk, because of the small amount of rotation that occurs between successive I/O requests. More specifically, because the disk platter continues rotating between requests, every sector that is requested will have just passed under the disk head and must wait a full rotation before passing again under the head. Depending on the rotation time, the performance can be worse than that of random I/O. However, if the drive supports command queuing, the application can issue write requests back-to-back. As such, there will always an I/O at the disk controller (i.e., no wasted rotations), and the application will see the full streaming bandwidth of the disk.

[1] For the inquiring mind, if there are n blocks in a storage system, the average jump distance from a given block i is $\sum_{k=1}^{n} \frac{|k-i|}{n}$. Therefore, the average over all n blocks is $\sum_{i=1}^{n} \frac{\sum_{k=1}^{n} \frac{|k-i|}{n}}{n}$, which simplifies to $\frac{n^2-1}{3n^2} \approx \frac{1}{3}$.

This above scenario also applies to sequential read requests when there is no read-ahead caching, though most modern storage systems provide read-ahead. In general, write-back and read-ahead caches provide performance benefits that are similar to command queuing. In effect, the storage system is issuing multiple outstanding requests (to/from these caches) on behalf of an application.

For the worse: multiple streams

Increasing concurrency by adding more streams often decreases performance. Consider the sequential I/O case just described. If one were to add a second sequential stream to a different region of the disk, the combined performance of the two streams could be much less than the streaming bandwidth of the disk. This is due to reduced efficiency (i.e., numerous seeks between the two streams) [21]. Although there are cases where combining streams actually improves efficiency, the streams must be carefully synchronized for this to occur. For example, if multiple processes are reading a file in an interleaved fashion (a common access pattern for scientific applications), combining the streams could improve storage system efficiency.

Characterizing concurrency

Because both types of concurrency (command queuing and multiple streams) have a potentially large impact on I/O performance, it would be ideal to characterize both. Unfortunately, working at the block level introduces a trade-off. Namely, it is difficult to distinguish among the different streams. Conversely, stream detection would be much easier if one were characterizing I/O at the file level (i.e., each process is a stream); but, again, one would somehow need to combine the characteristics of multiple streams and account for file system metadata, both of which would be difficult.

Given the goal of predicting overall storage system performance, characterizing the aggregate I/O workload at the block-level is desirable, even at the cost of losing information about the number of streams composing the workload. So, short of attempting stream detection (e.g., looking for sequential access, inspecting file system metadata to see which blocks belong to which files, etc.), the best one can do is to treat the entire file system as a single I/O stream. One can then approximate the multi-programming level of the file system by observing the number of outstanding commands in the storage system.

To calculate the average write queue depth, one can maintain a counter of write queue depths. When a new I/O is issued, the current write queue depth, which in this dissertation also includes any commands being executed, is added to the counter. To calculate the average write queue depth over a certain measurement window, one simply divides the counter by the number of write requests:

$$Write\ queue\ depth = \frac{Write\ queue\ counter}{Number\ of\ write\ requests} \tag{4.5}$$

The average read queue depth is calculated in the same manner.

4.1.5 Other workload characteristics

Many other workload characteristics have been found useful by other storage system researchers. The following examples illustrate, but they are not used in the evaluation of relative fitness in Chapter 7. This section can be skipped on a first reading of this dissertation.

Temperature

Data *temperature* is the ratio of the throughput to capacity [4]. For example, if a 1000 MB region of a disk is read at 1000 IO/sec, the data temperature is 1 IO/sec/MB. Of course, this measure does not take into account the spatial locality of the accesses (e.g., most of the accesses could be to the first block of the region). Nonetheless, it does give a basic measure of the locality of the I/O accesses.

Burst inter-arrival time

Information as to how often a stream is accessing data is useful. In particular, one would like to know the length of time between bursts of I/O. For example, it might be the case that an application computes for one hour and then enters an I/O phase. One can refer to the time spent computing as the "off" time and the time spent issuing I/O as the "on" time [1].

Of course, unless an I/O stream is asynchronous (open), and therefore unaffected by a storage system's service time, the "on" time can change across storage systems. For example, if a synchronous (closed) I/O stream with no think time (i.e., storage is the only bottleneck) completes in 60 seconds on device A, and device B is twice as fast as device A, then the same stream will complete in 30 seconds on device B — so the "on" time is variable. In effect, the "on" time is an indirect measure of the performance of a storage system and, therefore, only applies to the storage system on which the workload was characterized.

Stream phasing

The overlap of two streams is useful in predicting whether resource contention will affect performance. This is related to I/O concurrency. For example, if one stream is idle whenever the other stream is accessing data, there may not be as much conflict as when both streams are attempting to access data at the same time.

Although two streams can be overlapped in an infinite number of ways, an "overlap fraction" [1] can be used to quantify, in a course-grained fashion, how often two streams are issuing I/O concurrently. One could then use this fraction to estimate the reduction in performance due to extra seeks.

Temporal bursts

Previous research has shown that I/O requests typically come in bursts and are self-similar [9]. For example, suppose the "on" time of an I/O stream is 10 minutes. It may be the case that 80% of the accesses happen in the first 5 minutes. Also, when focusing on the first 5 minutes of I/O, it may be the case that the same 80/20 rule applies recursively (i.e., 80% of those accesses happen in the first 2.5 minutes). This is an example of self-similarity in the I/O stream, where one sees the same temporal burstiness, irrespective of the time scale. The "b-model" [24] can help describe such I/O behavior. It models temporal bursts in a way that is similar to the "80/20" law of databases (i.e., 80% of the accesses apply to 20% of the data).

Spatial-temporal correlation

It has been shown that disk traffic exhibits spatial burstiness [10] (i.e., some blocks are more popular than others) and that there is also a correlation with time [22] (i.e., certain blocks are accessed at certain times). The PQRS model [22] helps to describe the spatio-temporal correlations of such bursty traffic, by recursively subdividing space-time into regions. A value is assigned to each region, specifying the probability that a given region of blocks will be accessed at a given time. The PQRS model has been used to synthetically generate representative I/O streams of bursty workloads. It has also been used in predicting the performance of such workloads [23] using classification and regression trees (CART). There will be much more discussion of CART in Chapters 6 and 7.

I/O arrival rate

Some latency models will use a workload's I/O arrival rate as a workload characteristic [23]. However, this only applies to asynchronous workloads, where the I/O arrival rate is not dependent on the I/O service time in the storage system. For open workloads, the I/O arrival rate is, in queuing theoretical terms, a description of the I/O arrival process. For synchronous workloads, however, the arrival process is dependent on the service rate (i.e., faster storage means a faster application). Therefore, a closed workload's I/O arrival rate will not apply across different storage systems and, as such, cannot be used as a reliable workload characteristic.

4.1.6 Issues and challenges

There is a well-known challenge with workload characterization and another that is not so well-known. The first relates to compressing an I/O trace without losing information. The second relates to the volatility of certain workload characteristics. Specifically, as will be shown in Chapter 7, the block-level I/O characteristics of a synchronous workload can vary across storage systems.

Lossy compression

The goal of workload characterization is to compress an I/O stream into a set of characteristics that can be used describe the performance of a storage system for a particular workload. In doing so, however, there is the potential to lose performance-affecting information (e.g., correlations among characteristics, such as the spatio-temporal correlation). The desire for concise workload characteristics comes from their intended usage. As examples, one way to predict performance is to synthetically generate I/O that is representative of an application, and another is to use a performance model. Synthetic workload generation and performance modeling will be discussed in detail in Chapter 5. As might be expected, these tasks are much easier, though not always as accurate, when the workload characteristics are concise.

To better illustrate the trade-off between expressiveness and conciseness, suppose that a numerical average is used to characterize the I/O request size of a workload. Although an average is very concise, it is not very expressive. Indeed, there could be any number of workloads with different distributions of request size and the same average. To distinguish such workloads, quartiles could also be specified (i.e., the 25th, 50th, and 75th percentiles of request size), but at the cost of conciseness (now 4 number are

used in the characterization). Yet, still, some workloads could have the same percentiles and the same average, but a different range of request sizes. So, the minimum and maximum request sizes could also be specified. In general, one could keep adding information to the characterization until, eventually, the complete distribution of I/O request sizes has been specified (i.e., the I/O trace).

So, the big question is how much information to use. At the very least, workload characterization should include enough information to distinguish workloads that have significantly different performance. Stated simply, workloads with different performance should have different workload characteristics. Consider, for example, two files that are being accessed with the same read/write ratio, the same distribution of request sizes, the same level of concurrency, the same spatial access pattern (sequential), the same temporal access pattern, and so on. One would expect that the workloads would have similar performance. However, perhaps one file is being processed forward and the other backward. As a result, the performance of these two workloads could be very different (storage systems do not always perform well in reverse). That is, a key workload characteristic is missing.

However, even if the workload characteristics are expressive enough to distinguish workloads with different performance, there is no guarantee that they can be used to accurately synthesize an I/O stream or accurately predict performance using a model. In general, synthetic workload generators and performance models require only those workload characteristics that will affect storage system performance, and the granularity in which they are specified must match the sensitivity of the storage system (e.g., if a storage system is sensitive to 1 KB changes in the request size, the workload characteristics should be specified with a granularity of at least 1 KB — rounding to, say, the nearest 10 KB would hide information and may lead to inaccurate workload generation and poor performance predictions.) Determining the right amount of information to add to a set of workload characteristics is an unsolved problem, and has been for many years [7]. Usually, this involves iteratively testing a set of workload characteristics, using either a synthetic workload generator or a model, until all performance-affecting I/O behavior information has been adequately characterized (i.e., the characteristics can be used to generate synthetic I/O with the same performance as the characterized workload or used in a model to predict the performance of the real workload). This is a laborious process and one that is not easily automated.

The key (performance-affecting) workload characteristics may also depend on the storage system in question. To continue the above example, the "missing" workload characteristic may have gone unnoticed in a solid-state storage system (no moving parts), as reading a file in reverse may have the same performance as reading it forward. As another example, a storage system with a large cache may be less sensitive to the spatial access pattern of writes than would be a storage system with little cache. So, models of these systems would likely focus on different workload characteristics when predicting performance. Although there has been research on automatically distilling the key workload characteristics for a given workload and storage system [13], there must still be a rich set of workload characteristics from which the distillation can proceed (e.g., someone would have needed to discover that reading a file in reverse is a characteristics worth capturing). Moreover, to create a set of workload characteristics for any application, one would need to determine this superset *a priori*, distill the proper characteristics for the various storage systems one expects the workload to encounter, and specify which characteristics apply to which storage systems. Needless to say, this is a very difficult problem given the variety of workloads and storage systems in existence.

Synchronous workloads

A second, lesser known, challenge relates to the volatility of certain workload characteristics. In particular, the block-level I/O characteristics of a synchronous (closed-loop) workload can change as the workload is moved across storage systems, similarly to how the I/O arrival rate can change. Recall that an operating system is, in effect, a large speed-matching buffer between the applications and the storage system. As such, much of the optimization (e.g., disk request scheduling, read-ahead, and request coalescing) is in a feedback loop with the storage system. If the speed of the storage system changes, so too can the behavior of the OS. Workloads that are completely asynchronous (open-loop) are not affected by performance feedback, because, by definition, there is no closed loop influencing the arrival process of I/O requests. However, given that most applications operate, at least in part, in a closed fashion [8], accounting for changes in workload characteristics is essential when predicting their performance.

As discussed, the most obvious change for a synchronous workload is the I/O arrival rate: if a storage system completes the I/O faster, then an application or the OS are likely to issue I/O faster. However, other characteristics of the I/O stream can also change, such as the average request size, the spatial randomness, and the read/write ratio. Such effects occur when file systems, page caches and other middleware sit between an application and the storage system. Although the application may issue the same I/O, the characteristics of the I/O, as seen by the storage system, could change due to a variety of interactions between an operating system and storage system.

It is also often the case that the arrival process of read requests is determined by the application, where those of the write requests is determined by the file system (i.e., the application simply writes into an OS page cache). As such, the arrival process of the write requests can be more sensitive to the speed of the storage system. This will be discussed further in the Chapter 7 evaluation. For example, a slower storage system can result in a workload with larger inter-arrival times and larger write requests (due to request coalescing) when compared to the same workload running on a faster system.[2]

4.2 Concluding remarks

Workload characterization is intimately tied to performance analysis, modeling, and prediction. Years of research have produced numerous characteristics as well as synthetic workload generators and models to put them to good use. However, the challenges with workload characterization will likely get worse before they get better, as applications and storage systems continue to evolve. Relative fitness modeling is motivated by these challenges. As will be shown in Chapter 6, one can reduce the dependency on workload characteristics by modeling storage systems relative to another.

[2]The author once encountered this when benchmarking an iSCSI storage system [5]. Interestingly, for one of the I/O benchmarks, the performance of the storage system actually decreased when all I/O hit in the cache. Upon closer inspection, this was due to the extra latency of disk accesses, which gave the host OS more time to coalesce requests, thereby resulting in larger, more efficient (network-friendly) I/O.

Bibliography

[1] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: running circles around storage administration. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST 02)*, Monterey, CA, January 2002. The USENIX Association.

[2] Eric Anderson, Mahesh Kallahalla, Mustafa Uysal, and Ram Swaminathan. Buttress: A toolkit for flexible and high fidelity I/O benchmarking. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST 04)*, San Francisco, CA, March 2004. The USENIX Association.

[3] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP 91)*, Pacific Grove, CA, October 1991.

[4] Peter M. Chen and David A. Patterson. Storage Performance — Metrics and Benchmarks. *Proceedings of the IEEE*, 81(8):1151–1165, August 1993.

[5] Intel Corporation. Open Storage Toolkit. http://www.sourceforge.net/projects/intel-iscsi.

[6] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proceedings of the 2th USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March 2003. The USENIX Association.

[7] Gregory R. Ganger. Generating Representative Synthetic Workloads: An Unsolved Problem. In *Proceedings of the 21st International Computer Measurement Group Conference (CMG)*, Nashville, TN, December 1996. Computer Measurement Group (CMG).

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

[9] Maria E. Gomez and Vicente Santonja. Analysis of self-similarity in I/O workload using structural modeling. In *Proceedings of the 7th International Symposium on Modeling Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS-1999)*, College Park, MD, October 1999. IEEE Computer Society.

[10] Maria E. Gomez and Vicente Santonja. A new approach in the analysis and modeling of disk access patterns. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS 2000)*, Austin, TX, April 2000. IEEE.

[11] Maria E. Gomez and Vicente Santonja. A new approach in the modeling and generation of synthetic disk workload. In *Proceedings of the 8th International Symposium on Modeling Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS-2000)*, San Francisco, CA, August 2000. IEEE Computer Society.

[12] Kimberly Keeton, Guillermo A. Alvarez, Erik Riedel, and Mustafa Uysal. Characterizing Data-Intensive Workloads on Modern Disk Arrays. In *Proceedings of the 4th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-2001)*, Monterrey, Mexico, January 2001. IEEE.

[13] Zachary Kurmas, Kimberly Keeton, and Kenneth Mackenzie. Synthesizing Representative I/O Workloads Using Iterative Distillation. In *Proceedings of the 11th International Symposium on Modeling Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS-2003)*, Orlando, FL, October 2003. IEEE/ACM.

[14] John K. Ousterhout, Hervg Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2BSD File System. In *Proceedings of the 10th ACM Symposium on Operating System Principles (SOSP 01)*, Orcas Island, WA, December 1985.

[15] K. K. Ramakrishnan, Prabuddha Biswas, and Ramakrishna Karedla. Analysis of file i/o traces in commercial computing environments. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1992)*, Newport, RI, June 1–5 1992. ACM Press.

[16] Drew Roselli and Thomas E. Anderson. Characteristics of file system workloads. Technical Report UCB//CSD-98-1029, University of California, Berkeley. Computer Science Division, July 1998.

[17] Chris Ruemmler and John Wilkes. UNIX disk access patterns. In *Proceedings of the USENIX Winter 1993 Technical Conference*, San Diego, CA, January 1993. The USENIX Association.

[18] Mahadev Satyanarayanan. A study of file sizes and functional lifetimes. In *Proceedings of the 8th ACM Symposium on Operating System Principles (SOSP 81)*, Pacific Grove, CA, December 14–16 1981. ACM Press.

[19] Evgenia Smirni and Daniel A. Reed. Lessons from Characterizing the Input/Output Behavior of Parallel Scientific Applications. *ACM SIGMETRICS Performance Evaluation Review (PER)*, 33(1):27–44, June 1998.

[20] Alistair Veitch. The Rubicon workload characterization tool. Technical Report HPL-SSP-2003-13, Hewlett-Packard Laboratories, March 2003.

[21] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance Insulation for Shared Storage Servers. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 07)*, San Jose, CA, February 2007. The USENIX Association.

[22] Mengzhi Wang, Anastassia Ailamaki, and Christos Faloutsos. Capturing the Spatio-Temporal Behavior of Real Traffic Data. In *Proceedings of the IFIP WG 7.3 International Symposium on Computer Modeling, Measurement and Evaluation (Performance 2002)*, Rome, Italy, September 2002. IFIP.

[23] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R. Ganger. Storage device performance prediction with CART models. In *Proceedings of the 12th International Symposium on Modeling Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS-2004)*, Volendam, The Netherlands, October 2004. IEEE.

[24] Mengzhi Wang, Tara Madhyastha, Ngai Hang Chan, Spiros Papadimitriou, and Christos Faloutsos. Data Mining Meets Performance Evaluation: Fast Algorithms for Modeling Bursty Traffic. In *Proceedings of the IEEE 18th International Conference on Data Engineering (ICDE'02)*, San Jose, CA, February 2002. IEEE.

[25] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems (TOCS)*, 14(1):108–136, February 1996.

Chapter 5

Predicting storage performance

> "*Occurrences in this domain are beyond the reach of exact prediction because of the variety of factors in operation, not because of any lack of order in nature.*" Albert Einstein.

As discussed in Chapter 2, an important aspect of storage management is assigning applications to the proper storage systems, and performance predictions play a large role in making this assignment. Often, administrators (or automated management tools) want to predict the performance of a storage system before assigning it to an application, so as to guarantee the proper quality of service and to avoid the wasted time in configuring the wrong storage system. Predictions are useful in a number of different management scenarios, including the initial assignment of an application to a storage system, tuning or reconfiguring a storage system for an application (e.g., changing the RAID level), upgrading an application's storage system (e.g., by adding more memory or more disks), and moving the application to a completely different storage system.

Of course, running an application on a new, or reconfigured, storage system is the most accurate way of predicting its performance, and this technique is common among system administrators. However, this is not always feasible. In many cases, the application is not available for testing (e.g., it cannot be taken offline due to ongoing work) or is too complex to configure and run (e.g., a nuclear simulation). Also, consider the steps necessary to change the storage system of an application:

Step 1: Take the application offline.

Step 2: Backup the data.

Step 3: [Re-]configure the storage system.

Step 4: Format the storage (e.g., create a file system or database).

Step 5: Reload the application data from the backup.

Step 6: Restart the application and ensure the proper performance.

The above steps would be an afternoon project, at minimum, and if the selected storage system or its configuration was not the proper choice, the steps must be repeated. Obviously, such a trial-and-error approach is only feasibly for the smallest of data centers. Further, it is not practical within the context

of automated storage system design, where numerous possible assignments (e.g., 1000s) are considered before a final decision is made.

Fortunately, there are many techniques for predicting performance that do not require so much time. In particular, applications can be replaced with I/O traces, synthetic workload generators, or workload characteristics; and, storage systems can be replaced with models: simulation models, emulation models, analytical models, and statistical models. In addition to storage system design, such predictive techniques are also useful for storage architecture. As examples, a storage architect may wonder about the effects of a new cache eviction policy, a different bus topology, or the addition of a technology yet to be developed (e.g., phase changing memory). Using models of hypothetical systems, one can begin to answer such questions.

Of course, due to the complexity of modern-day storage systems, the variety of application workloads, and the numerous, poorly understood, interactions between workloads and storage systems (e.g., caching effects and workload interference), it is difficult to predict exactly the performance of a storage system for an arbitrary workload. Still, close estimates of storage system's performance are extremely useful, both in making proper allocation decisions in a data center and in evaluating architectural trade-offs.

The remainder of this chapter introduces each of these predictive techniques in greater detail. Section 5.1 discusses I/O trace collection and replay and Section 5.2 discusses synthetic workload generators. Both of these techniques will be used in the evaluation of relative fitness modeling in Chapter 7. Section 5.3 discusses simulation-based models and Section 5.4 discusses emulation-based models. These methods are presented as alternatives to analytical and statistical performance models. However, they are not typically used in optimization-based storage system design and, therefore, are not directly related to relative fitness modeling. These sections can be skipped on a first reading of this dissertation. Section 5.5 discusses analytical and statistical modeling. Particular attention is paid to the statistical modeling section, as it forms the foundation for relative fitness modeling.

5.1 Trace replay

I/O traces play a critical role in storage systems evaluation. They are captured through a variety of mechanisms [3, 7, 4, 31, 21], analyzed to understand the characteristics and demands of different applications, and used to guide the generation of representative I/O workloads. I/O traces can be replayed to mimic the I/O behavior of an application and, therefore, can be used to predict an application's performance. Often, traces are much easier to work with than actual applications, particularly when the applications are complex to configure and run or involve confidential data or algorithms.

Given an application, one can trace the I/O accesses at either the file or block level. Figure 5.1 illustrates. File traces can be captured by tracing the system calls between an application and the file system (e.g., `open()`, `close()`, `read()`, and `write()`). This can be done by using a system call tracing tool, such as the Unix `strace` program, or through library interposition. For file-based tracing, one I/O trace is usually generated for each process (or thread) of interest. For block-based tracing, one trace is collected per storage device or, equivalently, a single trace is collected with device IDs associated with each I/O. Block-based tracing is accomplished by instrumenting either the block-device driver that an OS uses to access a storage system (e.g., a SCSI device driver) or by instrumenting the storage system.

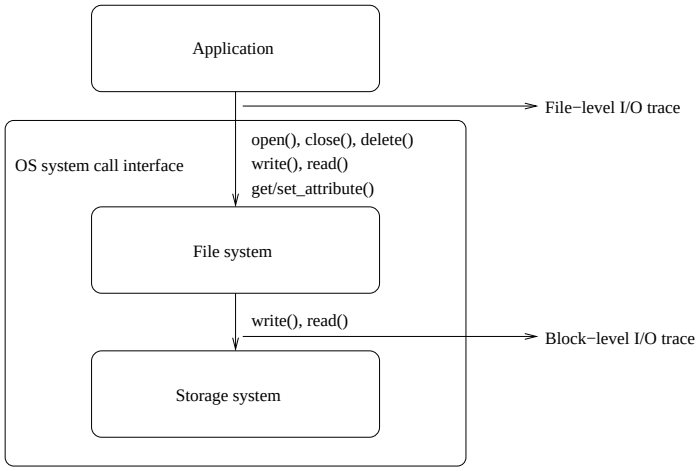

Figure 5.1: Trace collection can occur at the file or block level. File-level traces contain the I/O calls between an application and a file system. Block-level traces contain the I/O calls between a file system, or any block-based application, and a storage system.

After traces have been collected, user-level applications can be used to replay the I/O. Figure 5.2 illustrates. In the case of file-based replay, the system calls can be replayed almost as-is, using the same system call arguments that were captured in the trace. One needs only ensure that the files that existed prior to the trace collection, if any of these files are accessed by the traced application, are created before the replay begins. In addition, the file handles present in the traced system calls (e.g., those use in traced `write()` and `read()` calls) will need to be changed to reflect the file handles returned by other replayed calls, in particular the `open()` call. A block-based trace, after similarly slight modification, can be replayed directly against a storage system. In particular, one must map the traced device IDs to a new set of storage devices (or storage systems) and also ensure that the capacity of each device is sufficient to satisfy the offsets in the trace. In short, replaying I/O (without regard to timing) is relatively straightforward.

For performance prediction, however, trace replay usually occurs on a different storage system than that which was traced. Indeed, a trace replay provides little new information, if it occurs on the same system from which the trace was collected. Instead, one typically wants to predict the performance of a new storage system (real, simulated, or emulated). To do so, one must ensure the proper timing during I/O replay.

The biggest challenge with I/O trace replay, whether file-based or block-based, is replaying I/O at the right time. A replayer should issue I/Os at the same times as the application would, but this requires scaling I/O inter-arrival times with the speed of a storage system. There are two things that make this challenging: computation (e.g., application processing between each I/O) and data dependencies. Such information is not present in a traditional I/O trace. However, with a mechanism for automatically extracting such information from an application [21], a replayer can more closely mimic an application's I/O behavior.

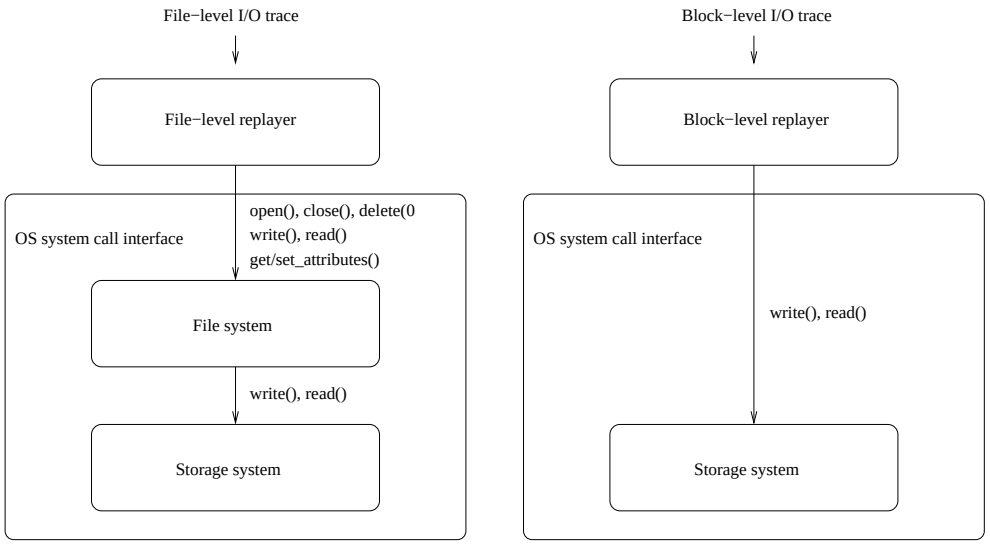

Figure 5.2: Trace replay can occur at the file level or block level. A file-level replay executes on top of a file system. A block-level replay executes on top of a raw storage system.

5.2 Synthetic I/O

Synthetic workload generators[3, 6, 11, 12] are programs that generate and issue I/O requests based on a specified set of workload characteristics. Typically, a synthetic workload generator runs as a user-level program and can be configured to issue I/O to a file or directly to a storage system. Figure 5.3 illustrates.

As discussed in Chapter 3, such workload generators can be used as micro-benchmarks to test specific types of workloads, using any of the workload characteristics described in Chapter 4. However, they can also be used to mimic the I/O behavior of a particular application and, therefore, used to predict the performance of various workloads.

The biggest advantage of synthetic I/O is its ease of use. Different workloads can be synthesized by varying the input parameters of the workload generator. The biggest challenge is synthesizing workloads that are difficult to characterize (e.g., applications with irregular temporal access patterns). In this regard, as discussed in Chapter 4, workload characterization and synthetic workload generation are two sides of the same coin — better workload characteristics improve the synthetic workloads [8]. In fact, a common way to measure the expressiveness of a new workload characteristic is to use the characteristic to generate synthetic I/O and to then compare the synthetic I/O to that of the real application from which the workload characteristic was obtained [14].

5.3 Simulated storage systems

Simulated storage systems model the behavior of real (or hypothetical) storage systems through the use of modules [5, 9, 23, 25, 29, 30]. Modules simulate various hardware system components, such as processors, caches, memory, I/O interconnects, and disks. Similarly, modules are used to simulate software functionality like cache eviction, I/O scheduling, and read-ahead. With rules that describe how

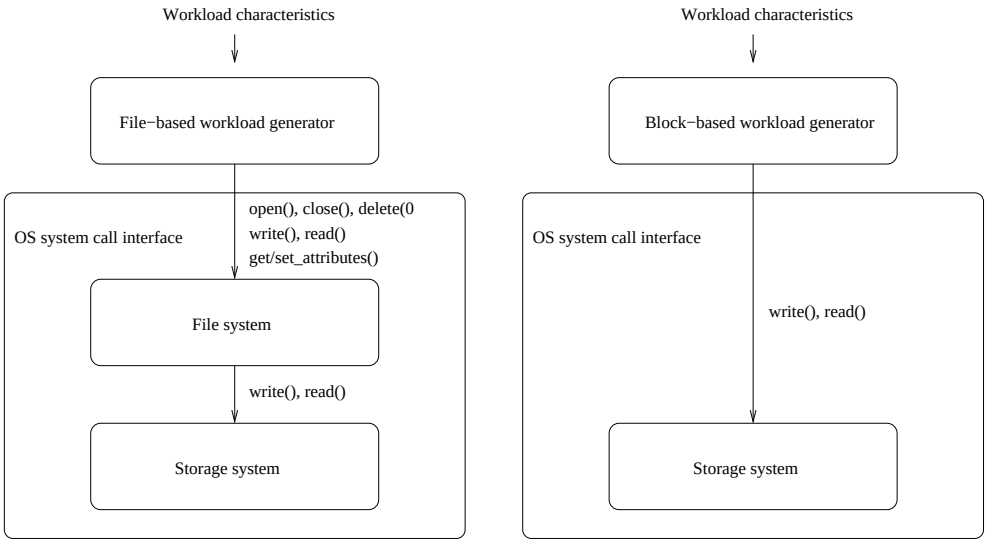

Figure 5.3: Synthetic workload generators can be configured to issue I/O into a file (left half of figure) or directly into a storage system (right half). The input parameters into a workload generator are the I/O characteristics of a workload.

modules are to be plugged together, a storage system designer can simulate different storage systems and evaluate them with various workloads. Figure 5.4 illustrates how one might simulate an entire RAID array by composing various simulation modules.

Because a simulated storage system is not a real storage system, but simply a model of how the real storage system would behave, it cannot be used (stand-alone) to host real applications. Instead, simulators either provide a programming environment for internally generating synthetic I/O or take an I/O trace as input. Simulators can be run in a timing-accurate mode, such that I/O requests take the same amount of wall-clock time as they would for the system being simulated. Simulators can also be run as fast as possible. This second option is useful if the goal is to locate system bottlenecks, where the relative performance of the simulated components is all that is needed.

The biggest advantage of a simulated storage system is the wide flexibility and the ability to explore various system designs. The biggest disadvantage is the considerable amount of time and expertise necessary to create an accurate simulation, as doing so requires detailed systems component models and calibration with a real system.

5.4 Emulated storage systems

In contrast to a simulator, an emulated storage system can be used to host real applications. It behaves as a real (or hypothetical) storage system would in terms of performance, but not necessarily in terms of persistence or reliability. Internally, emulators contain performance simulation models of the storage system in question, as previously described, such that a timing-accurate servicing of each I/O request is possible. Figure 5.5 illustrates. For example, emulation models of MEMS-based storage, a type of solid-state storage, have used DRAM and disks in place of an actual MEMS device [10].

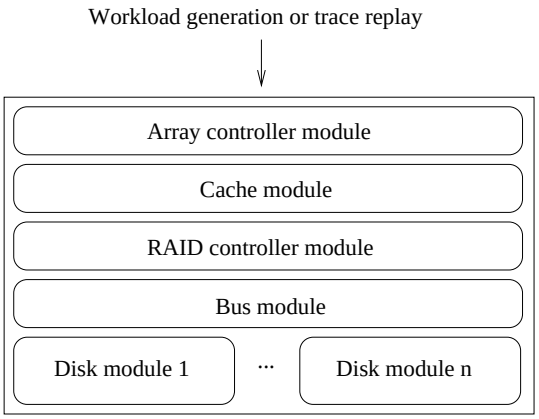

Figure 5.4: A RAID array might be simulated using modules for the front-end array controller, the cache, the back-end RAID controller, the system bus to which the disks are attached, and the disk drives. Such a model is driven using synthetic workloads or I/O trace replay.

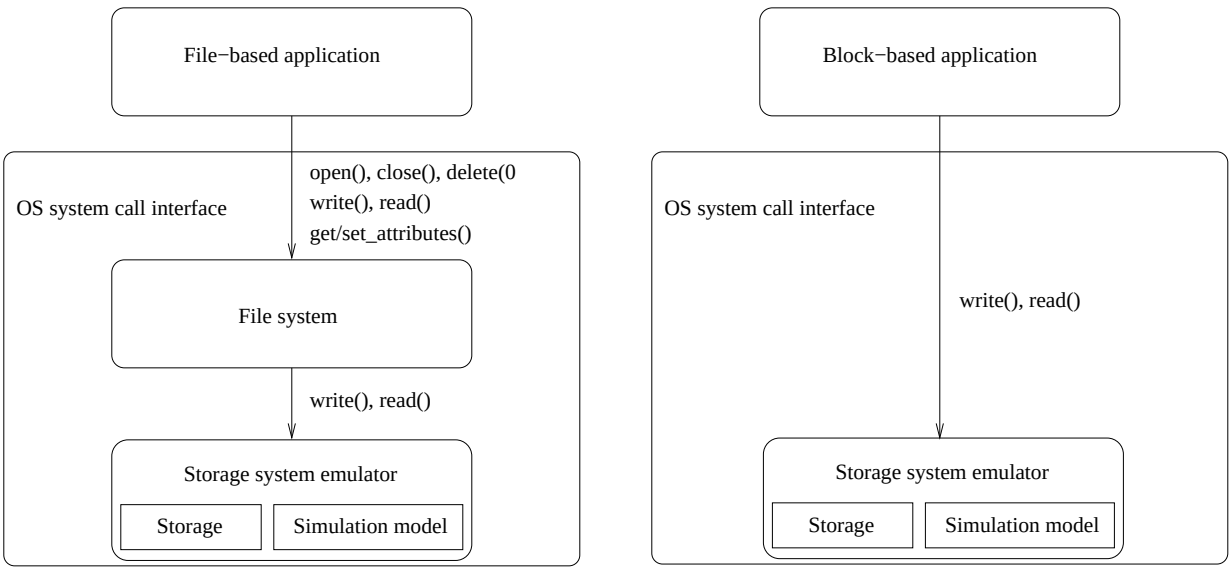

Figure 5.5: A storage system emulator can host real file-based applications (left half of figure) or block-based applications (right half). Internally, an emulation-based model contains a simulation model of a real (or hypothetical) storage system. In addition, an emulation model contains some type of storage (e.g., perhaps just DRAM) for storing and retrieving actual application data.

The biggest advantage of emulation is that an application can often run unmodified. Such is the case when the emulation model implements a standard storage interface (e.g., IDE or SCSI). The application's performance can then be analyzed (e.g., monitored, traced, or characterized) as though it were running on a real storage system. The disadvantages of emulation models are the same as those of simulation models. Namely, significant expertise is required to make them accurate.

5.5 Analytical and statistical models

Whereas simulated or emulated storage systems require a workload as input, an analytical or statistical model only requires an application's workload characteristics. Such models can be further described as either being white-box (requiring of knowledge of a storage system's internal structure) or black-box (requiring no knowledge). Given the challenges in modeling modern-day (complex) storage systems [27], black-box approaches are becoming an attractive alternative [1, 13, 23, 28]. Figure 5.6 illustrates how an I/O workload can be traced and characterized, and how the characteristics can later be input into the models of various storage systems.

Because of their speed, these models are often preferred in the context of automated storage system design, as described in Chapter 2, where numerous predictions are potentially requested by an optimization solver [16, 2] before a final management decision is made. In contrast, simulated or emulated storage systems must take a measurement over a certain period of time whenever a performance prediction is needed, just as one would when measuring the performance of a real storage system.

Conventionally, for a given storage system, one analytical or statistical model is built for each performance metric of interest. Further, the workload characteristics used by the models are assumed to be "absolute," as described in Chapter 4. That is, any of the storage systems can be used to characterize a workload, after which the workload characteristics can be input into any of the models to make predictions. However, as discussed, this can lead to problems for synchronous workloads, where the block-level I/O characteristics can depend on the underlying storage system performance.

The remainder of this chapter provides a more detailed overview of conventional analytical and statistical performance models, discussing their challenges and continuing to build the case for relative fitness modeling.

5.5.1 Analytical models

Analytical models use mathematical constructs to calculate (predict) the performance of a storage system based on the characteristics of a workload [15, 17, 18, 19, 16, 24, 26, 27]. For example, one simple model of a disk's throughput, for randomly accessed data, is the inverse of its average seek time:

$$Throughput = \frac{1}{Average\ seek\ time} \tag{5.1}$$

To calibrate such a model for any given disk drive, one would measure the drive's average seek time (a modeling parameter). For example, if the average seek time of a disk is 6 ms, then the predicted average

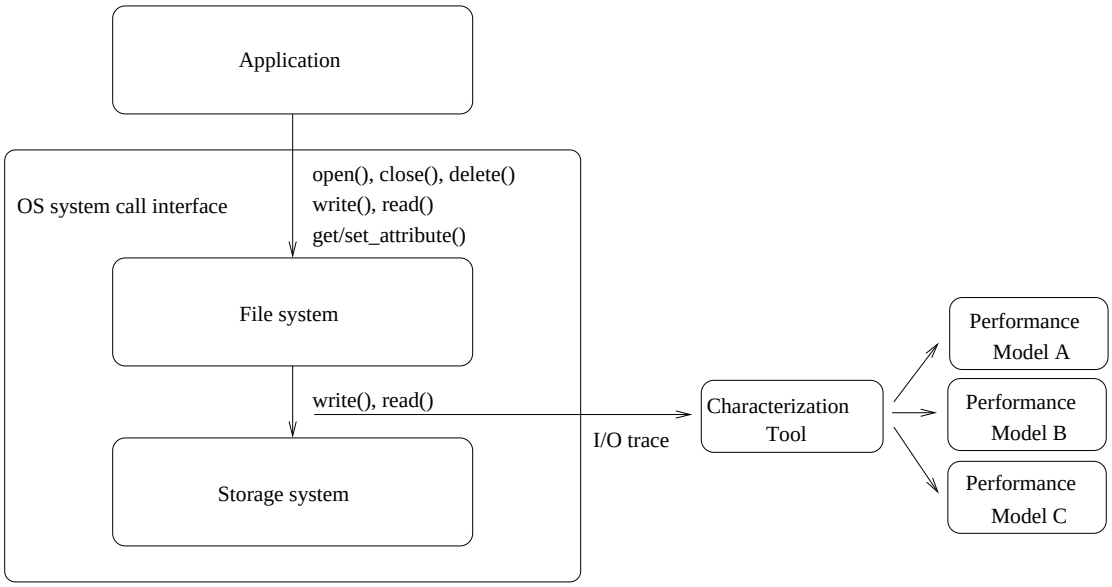

Figure 5.6: Analytical or statistical performance models accept workload characteristics as input. The characteristics can be obtained by analyzing an I/O trace (offline or online). Though the figure illustrates block-level I/O, models could also be built to predict file-level performance.

throughput is 1/6 ms, or 167 IOPS. This model requires no workload characteristics as input and always predicts a throughput of 167 IOPS.

As a model builder, one might discover that prediction accuracy can be improved by modeling the rotational delay and the data transfer time of each I/O request. Specifically, a random I/O request waits, on average, one half of a rotation before the first desired sector arrives at the disk head for reading, and the transfer time depends on the disk's data transfer rate and the size of the I/O request (a workload characteristic). As such, the model can be improved as follows:

$$Throughput = \frac{1}{Average\ seek\ time + \frac{Rotation\ time}{2} + \frac{Transfer\ rate}{Request\ size}} \tag{5.2}$$

Though simplistic, the above example captures the essence of how an analytical model is constructed, calibrated against a real system (e.g., the seek time, rotation time, and transfer rate), and used with workload characteristics (e.g., the request size). In the literature, analytical models are much more complex. They model the performance of disk drives over a variety of workload characteristics, like those discussed in Chapter 4. They have also been extended to model RAID controllers, caches, and many optimizations often found in a storage system (e.g., write-back caching and read-ahead).

The biggest advantages of analytical models, over simulation models, are their speed and ease-of-use. The only input is a description of the workload, and the output is a performance prediction. As discussed in Chapter 2, such models are preferred in automated storage system design, as they can be quickly invoked by optimization solvers.

However, there are two big challenges with analytical modeling. First, one must model the complex

interactions of various storage system components and do so in a way that is mathematically tractable to the model builder. Second, an analytical model is dependent on the quality of the workload characteristics used to describe the I/O. And, as discussed in Chapter 4, these challenges often play against one another. Specifically, mathematical tractability argues for more concise characteristics (e.g., numbers that can be plugged into equations), but often at the cost of a reduction in the quality of the workload characteristics.

5.5.2 Statistical models

Statistical models use the performance of past workloads to predict that of future workloads [1, 13, 20, 22, 28]. Perhaps the simplest of all black-box models is a numerical average. For example, given the performance of a collection of random I/O workload samples on a disk drive with a 6 ms average seek time, a statistical model could learn that random accesses, on average, perform at 167 IOPS. However, just as with analytical models, statistical models are more accurate when they use workload characteristics.

Table 5.1, for example, shows a simplified table-based model [1], recording the average bandwidth of a disk drive for various request sizes. Of course, some form of interpolation is required when an exact match is not found in the table. To predict the performance of, say, a 3 KB request, one might average the 2 KB and 4 KB performance and predict 37 MB/sec.

Figure 5.7 shows the same information modeled with a classification and regression tree (CART), another type of statistical model that can be used to predict the performance of storage systems [28]. In this example, to predict the performance of a 3 KB request size, one would follow the correct path in the tree until a leaf node is reached — 42 MB/sec, in this case.

Like analytical models, statistical models are fast and easy to use. The biggest advantage over an analytical model is the ability to *learn* the complex interactions of a storage system, by example, rather than attempting to model them through equations. This is especially advantageous in the context of black-box modeling, where the internal workings of a storage system are unknown to the model builder. A disadvantage, like analytical models, is a dependence on workload characteristics. Unlike analytical models, statistical models also require a potentially large amount of training data, so as to establish a strong statistical relationship between workload characteristics and storage system performance.

5.5.3 Limitations of absolute models

In this work, conventional performance models (analytical or statistical) are referred to as absolute models, both to distinguish them from the relative fitness models presented in the next chapter and to stress that the workload characteristics used by the models are assumed "absolute" by the model builders (i.e., not relative to the storage system from which they are obtained).

More formally, an absolute model of a storage system i is any function F_i that maps an application's workload characteristics $\mathbf{WC}_i$ to a performance prediction P_i, where $\mathbf{WC}_i$ is a vector of values representing workload characteristics, and P_i is any performance metric (e.g., bandwidth, throughput, and latency). This relationship be expressed as follows:

$$\mathrm{P}_i = F_i(\mathbf{WC}_i) \tag{5.3}$$

Request size	Bandwidth
1 KB	15 MB/sec
2 KB	27 MB/sec
4 KB	42 MB/sec
8 KB	66 MB/sec

Table 5.1: A table-based model that records the performance of a disk drive for sequentially-read data.

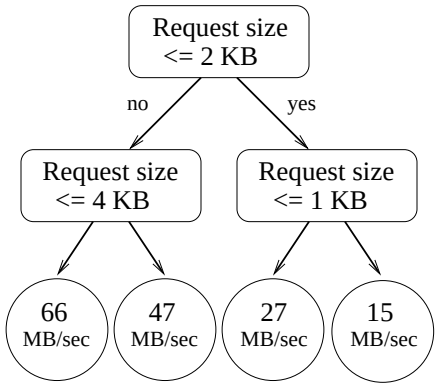

Figure 5.7: A regression tree that learns the performance of a disk drive for sequentially-read data.

The key insight behind Equation 5.3 is that an absolute model assumes that it is using the characteristics of a workload running on storage system i to predict the performance of storage system i. However, such an idealized scenario will not occur in practice, as obtaining workload characteristics from storage system i requires one to first run the corresponding application on that storage system. Of course, if one runs an application, there is no need for a performance prediction. The performance can be observed.

Therefore, in practice, an absolute model is used to predict the performance of moving a workload from some storage system j to another storage system i. This involves characterizing the workload on storage system j to obtain workload characteristics $\mathbf{WC}_j$ and then inputting $\mathbf{WC}_j$ into the performance model of storage system i. However, as will be shown in the Chapter 7 evaluation, many of the presumably "absolute" workload characteristics can change across storage systems, leading to additional prediction error. In other words, it is often the case that $\mathbf{WC}_j \neq \mathbf{WC}_i$, and the difference matters.

To illustrate how changing workloads characteristics can be problematic, consider the case where the average request size of a workload differs between two storage systems (e.g., because of a difference in request coalescing in the OS). Specifically, suppose the average request sizes for a given workload on storage systems j and i are 1 KB and 2 KB, respectively. If the workload characteristics measured on storage system j are input into the performance model for storage system i, the predicted performance (referring back to Table 5.1 or Figure 5.7) is 15 MB/sec instead of the actual performance of a 2 KB request on storage system i, which is 27 MB/sec. This hypothetical example illustrates the risk of indexing into a statistical model with inaccurate workload characteristics. The same misprediction can occur with an analytical model, as one is making a calculation with an incorrect workload characteristic.

5.6 Summary and concluding remarks

The need for accurate performance predictions has given rise to a variety of predictive techniques. One can model just the application (using trace replay or synthetic I/O), just the storage system (using simulation or emulation), or both the application and the storage system (using workload characteristics and analytical or statistical performance models).

As discussed in Chapter 2, one of the goals of an automated storage system is to assign workloads to the proper storage. Doing so requires exploring a number of possible assignments within the context of an optimization solver and, therefore, requires that prediction time be quick. As such, analytical or statistical models are the ideal choice, and black-box modeling is especially attractive given the increasing complexity of storage systems.

For better, and for worse, workload characterization sits at the core of performance modeling. It makes the modeling both possible and challenging. As discussed in Chapter 4, numerous workload characteristics have been developed to help explain and predict the performance of storage systems. However, not all applications lend themselves to characterization, especially those with irregular patterns of access, and not all storage systems are concerned with the same workload characteristics. Further, the workload characteristics of synchronous I/O, due to the performance feedback of a closed loop, can change when the workload is moved from one storage system to another.

Collectively, the above challenges have presented a major roadblock to analytical or statistical performance modeling becoming a widespread and dependable tool for the data center, and they are the motivation for relative fitness modeling. As will be shown in the following chapters, by assuming that workload characteristics are *relative* to a specific storage system, one can build models between pairs of storage systems and use the performance of one storage system to predict that of another, thereby reducing the dependency on workload characteristics. In addition, relative modeling implicitly learns how workload characteristics can change across storage systems.

Bibliography

[1] Eric Anderson. Simple table-based modeling of storage devices. Technical Report HPL-SSP-2001-4, Hewlett-Packard Laboratories, July 2001.

[2] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: running circles around storage administration. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST 02)*, Monterey, CA, January 2002. The USENIX Association.

[3] Eric Anderson, Mahesh Kallahalla, Mustafa Uysal, and Ram Swaminathan. Buttress: A toolkit for flexible and high fidelity I/O benchmarking. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST 04)*, San Francisco, CA, March 2004. The USENIX Association.

[4] Akshat Aranya, Charles P. Wright, and Erez Zadok. Tracefs: A File System to Trace Them All. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST 04)*, San Francisco, CA, March 2004. The USENIX Association.

[5] John Bucy and Gregory R. Ganger et al. The DiskSim Simulation Environment Version 3.0 Reference Manual. Technical Report CMU-CS-03-102, Carnegie Mellon University, January 2003.

[6] Intel Corporation. Open Storage Toolkit. http://www.sourceforge.net/projects/intel-iscsi.

[7] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proceedings of the 2th USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March 2003. The USENIX Association.

[8] Gregory R. Ganger. Generating Representative Synthetic Workloads: An Unsolved Problem. In *Proceedings of the 21st International Computer Measurement Group Conference (CMG)*, Nashville, TN, December 1996. Computer Measurement Group (CMG).

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

[10] John L. Griffin, Jiri Schindler, Steven W. Schlosser, John S. Bucy, and Gregory R. Ganger. Timing-accurate storage emulation. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST 02)*, Monterey, CA, January 2002. The USENIX Association.

[11] Intel Corporation. Iometer. http://www.iometer.org.

[12] I/O Performance Inc. Xdd. http://www.ioperformance.com.

[13] Terence Kelly, Ira Cohen, Moises Goldszmidt, and Kimberly Keeton. Inducing Models of Black-Box storage Arrays. Technical Report HPL-2004-108, Hewlett-Packard Laboratories, June 2004.

[14] Zachary Kurmas, Kimberly Keeton, and Kenneth Mackenzie. Synthesizing Representative I/O Workloads Using Iterative Distillation. In *Proceedings of the 11th International Symposium on Modeling Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS-2003)*, Orlando, FL, October 2003. IEEE/ACM.

[15] Edward K. Lee and Randy H. Katz. An analytic performance model of disk arrays. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1993)*, Santa Clara, CA, May 1993. ACM Press.

[16] Arif Merchant and Guillermo A. Alvarez. Disk array models in minerva. Technical Report HPL-2001-118, Hewlett-Packard Laboratories, April 2001.

[17] Arif Merchant and Philip S. Yu. An analytical model of reconstruction time in mirrored disks. *ACM SIGMETRICS Performance Evaluation Review (PER)*, 20(1-3):115–129, May 1994.

[18] Arif Merchant and Philip S. Yu. Analytic modeling and comparisons of striping strategies for replicated disk arrays. *IEEE Transactions on Computer Systems*, 44(3):419–33, March 1995.

[19] Arif Merchant and Philip S. Yu. Analytic modeling of clustered raid with mapping based on nearly random permutation. *IEEE Transactions on Computer Systems*, 45(3):367–373, March 1996.

[20] Michael Mesnier, Brandon Salmon, Matthew Wachs, and Gregory Ganger. Relative fitness models for storage. *ACM SIGMETRICS Performance Evaluation Review (PER)*, 33(4):23–28, June 2006.

[21] Michael Mesnier, Matthew Wachs, Raja R. Sambasivan, Julio Lopez, James Hendricks, Gregory R. Ganger, and David O'Hallaron. //TRACE: Parallel Trace Replay with Approximate Causal Events. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 07)*, San Jose, CA, February 2007. The USENIX Association.

[22] Michael Mesnier, Matthew Wachs, Raja R. Sambasivan, Alice Zheng, and Gregory R. Ganger. Modeling the relative fitness of storage. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2007)*, San Diego, CA, June 2007. ACM Press.

[23] Florentina I. Popovici, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Robust, portable I/O scheduling with the disk mimic. In *Proceedings of the 2003 USENIX Annual Technical Conference*, San Antonio, TX, June 2003. The USENIX Association.

[24] Elizabeth Shriver, Arif Merchant, and John Wilkes. An analytic behavior model for disk drives with readahead caches and request reordering. *ACM SIGMETRICS Performance Evaluation Review (PER)*, 26(1):182–191, June 1998.

[25] Eno Thereska, Michael Abd-El-Malek, Jay J. Wylie, Dushyanth Narayanan, and Gregory R. Ganger. Informed Data Distribution Selection in a Self-predicting Storage System. In *Proceedings of the 3rd IEEE International Conference on Autonomic Computing (ICAC-06)*, Dublin, Ireland, June 2006. IEEE.

[26] Mustafa Uysal, Guillermo A. Alvarez, and Arif Merchant. A modular, analytical throughput model for modern disk arrays. In *Proceedings of the 9th International Symposium on Modeling Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS-2001)*, Cincinnati, OH, August 2001. IEEE/ACM.

[27] Elizabeth Varki, Arif Merchant, and Jianzhang Xu ndXiaozhou Qiu. Issues and challenges in the performance analysis of real disk arrays. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 15(6):559–574, June 2004.

[28] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R. Ganger. Storage device performance prediction with CART models. In *Proceedings of the 12th International Symposium on Modeling Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS-2004)*, Volendam, The Netherlands, October 2004. IEEE.

[29] John Wilkes. The Pantheon storage-system simulator. Technical Report HPL-SSP-95-14, Hewlett-Packard Laboratories, December 1995.

[30] Bruce Worthington, Gregory R. Ganger, Yale Patt, and John Wilkes. On-line extraction of SCSI disk drive parameters. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1995)*, Ottawa, Canada, May 1995. ACM Press.

[31] Ningning Zhu, Jiawu Chen, and Tzi-Cker Chiueh. TBBT: Scalable and Accurate Trace Replay for File Server Evaluation. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST 05)*, San Francisco, CA, December 2005. The USENIX Association.

Chapter 6

Relative fitness modeling

Relative fitness: the fitness of a genotype compared with another in the same gene system [2].

Relative fitness modeling is a new statistical, black-box approach to modeling the performance of storage systems, motivated by the challenges with workload characterization and absolute modeling presented in the previous chapters. Rather than attempt to discover new workload characteristics that are expressive, yet concise, and "absolute" across storage systems, the goal of relative fitness modeling is to use observed performance and resource utilization, in addition to workload characteristics, to predict performance. Of course, doing so requires one to abandon the notion of an absolute model, as performance and resource utilization are specific to a given storage system.

The insight behind relative fitness is best obtained through analogy. When predicting your performance in a college course (a useful prediction during enrollment), it is helpful to know the grade received by a peer and the number of hours he worked each week to achieve that grade (his resource utilization). Naturally, our own performance for certain tasks is a complex function of the characteristics of the task and our ability. However, we often learn to make predictions relative to the experiences of others, because it is easier.

Applying the analogy, two complex storage systems may be reasonable predictors for one another. For example, they may have similar RAID levels, caching algorithms, or hardware platforms. As such, their performance may be related. Even dissimilar storage systems may be related in certain ways (e.g., for a given workload type, one usually performs well and the other poorly). The objective of relative fitness modeling is to learn these types of relationships. More specifically, relative fitness models learn to predict performance ratios rather than performance itself. For example, storage system A may be 30% faster than storage system B for small random reads, 40% faster for large sequential writes, and the same for all other workloads. Thus, only three relative fitness values are needed to describe the performance of storage system A relative to storage system B: 1.3, 1.4, and 1.0. Then, to predict the performance of storage system A, one simply multiplies the appropriate predicted performance ratio by the observed performance of storage system B.

Despite the many differences, relative fitness modeling can be regarded as an evolution of absolute modeling and is presented as such in this chapter. The differences between an absolute and relative fitness model are summarized here:

1. One absolute model is built for each storage system, for each performance metric of interest (e.g., bandwidth, throughput, or latency). In contrast, two relative fitness models are built for each pair of storage systems: one to predict the performance of storage system A given a workload running on storage system B, and vice versa.

2. An absolute model is trained to predict the performance of a storage system A using workload characteristics obtained from the same. In contrast, a relative fitness model is trained to predict the performance of storage system A using workload characteristics from a different storage system B. However, the usage of the absolute and relative fitness model is the same. To predict the performance of storage system A, one inputs the workload characteristics from a different storage system B (i.e., where the workload is currently running) into the performance model of storage system A.

3. An absolute model discards the observed performance and resource utilization of storage system B when making performance predictions for storage system A, because it is not trained to use this information. In contrast, a relative fitness model uses the observed performance and resource utilization of storage system B to predict the performance of storage system A.

4. An absolute model predicts performance values (e.g., that the bandwidth of a storage system A will be 30 MB/sec). In contrast, a relative fitness value predicts a performance ratio (e.g., that storage system A will be 50% faster than storage system B for the workload in question).

The rest of this chapter is organized as follows. Section 6.1 provides a formal definition of relative fitness modeling. Section 6.2 discusses how relative fitness modeling addresses many of the challenges faced by absolute modeling. Section 6.3 describes how to train a relative fitness model. Section 6.4 shows how to build a relative fitness model with a classification and regression tree (CART). Section 6.5 discusses the cost of relative fitness modeling and concludes the chapter.

6.1 Derivation of relative fitness

Relative fitness begins with an absolute model (Eq. 5.3). Recall that a workload is running on storage system j, $\mathbf{WC}_j$ is measured on storage system j, and the performance of storage system i is to be predicted using the absolute model F_i.

The first objective of relative fitness is to capture the changes in workload characteristics from storage system j to i, that is, to predict $\mathbf{WC}_i$ given $\mathbf{WC}_j$. Such change is dependent on the storage systems, so a function $G_{j \to i}$ is defined to predict how the workload characteristics will change from j to i:

$$\mathbf{WC}_i = G_{j \to i}(\mathbf{WC}_j).$$

$G_{j \to i}$ can now be used in the context of the absolute model F_i to predict the performance of storage system i:

$$\mathrm{P}_i = F_i(G_{j \to i}(\mathbf{WC}_j)).$$

However, rather than learn two functions, the composition of F_i and $G_{j \to i}$ can be expressed as a single composite function $RM_{j \to i}$, called a *relative model*:

$$\mathrm{P}_i = RM_{j \to i}(\mathbf{WC}_j). \tag{6.1}$$

With each model now involving an origin j and target i, one can use the performance of storage system j ($\mathbf{Perf}_j$) and its resource utilization ($\mathbf{Util}_j$) to help predict the performance of storage system i. $\mathbf{Perf}_j$ is a vector of performance metrics such as bandwidth, throughput and latency. $\mathbf{Util}_j$ is a vector of utilization metrics such as storage system j's cache utilization, network utilization, and CPU utilization:

$$\mathrm{P}_i = RM_{j \to i}(\mathbf{WC}_j, \mathbf{Perf}_j, \mathbf{Util}_j). \tag{6.2}$$

Equation 6.2 is called a *relative performance model*, to reflect the fact that performance (and utilization) are used in making the predictions. That is, one can now describe a workload in terms of its effects on a given storage system. Note that with absolute modeling, only one model is trained for each storage system, thereby precluding the use of performance and resource utilization, which are specific to a given storage system. Including such information in an absolute model would require that one specify the system from which the performance and utilization metrics were obtained, and this is equivalent to relative modeling.

Next, rather than predict performance values, one can predict performance ratios. The expectation is that ratios are better predictors for new workloads, as they naturally interpolate between known training samples. Such a model is called a *relative fitness model*:

$$\frac{\mathrm{P}_i}{\mathrm{P}_j} = RF_{j \to i}(\mathbf{WC}_j, \mathbf{Perf}_j, \mathbf{Util}_j). \tag{6.3}$$

To use the relative fitness model, one solves for P_i:

$$\mathrm{P}_i = RF_{j \to i}(\mathbf{WC}_j, \mathbf{Perf}_j, \mathbf{Util}_j) \times \mathrm{P}_j.$$

Models based on Equations 5.3 through 6.3 are evaluated in Chapter 7. The relative model that only uses workload characteristics (Eq. 6.1) is included to separate the benefits of modeling changes in the workload characteristics from that of using performance to make predictions (Eqs. 6.2 and 6.3).

6.2 Discussion

By modeling storage systems relative to one another, relative fitness models address many of the challenges associated with absolute modeling. Specifically, the dependence on workload characteristics is reduced, the feedback between a synchronous workload and a storage system is implicitly modeled, and models can potentially be trained with less data. Each of these points is discussed in more detail below.

First, unlike absolute models, relative fitness models allow the use of performance and resource utilization when making predictions, thereby reducing the dependency on workload characteristics. For

example, rather than attempt to describe the spatio-temporal correlation of an I/O stream (a characteristic that could be useful in predicting a workload's cache hit rate and, hence, its performance), one can simply use the cache hit rate of one storage system when predicting the performance of another storage system. Similarly, the average request latency of a workload gives some indication of the effectiveness of a storage system's cache and, therefore, is an indirect measure of cache locality. With relative fitness models, both of these characteristics can be used to help predict performance. In general, performance and resource utilization are much easier to obtain than workload characteristics and, as will be shown in the next chapter, are usually better predictors of performance.

Second, because relative fitness models are constructed between every pair of storage systems, the feedback of a synchronous workload is implicitly modeled as it moves from one storage system to another. As such, changes in the workload characteristics will not affect prediction accuracy like they would for an absolute model. Recall the example from Chapter 5, where the request size changed from 1 KB on storage system B to 2 KB on storage system A, resulting in a misprediction. With a relative fitness model, it is of no consequence that these characteristics change between storage systems B and A, because the performance of storage system A is modeled relative to the characteristics on storage system B, not A.

Third, relative fitness models help address a general problem with statistical modeling: the need for a potentially large amount of training data. By predicting performance ratios rather than performance values, a relative fitness model can often learn the performance of a storage system with fewer training samples. In effect, a relative fitness model predicts linear functions (the predicted performance ratio is the slope in a line that relates the performance of storage system A to storage system B) where an absolute model predicts constant values. Such linear functions can be learned with a simple piecewise-constant regression model rather than a more complex piecewise-linear model [3].

To illustrate the benefit of performance ratios, by analogy, suppose that Sally always does 10% better than Bill on exams and that we want to train a model to learn this. On one specific exam, Bill gets 20 points and Sally gets 22. Using this exam as training data, a relative fitness model will learn the performance ratio of 1.10 between Bill and Sally. In contrast, an absolute model for Sally will simply learn that she got 22 points on the exam. Now, if Bill took another exam and got 50 points, the relative fitness model would predict a performance ratio of 1.10, resulting in a prediction of 55 for Sally, and the absolute model would simply predict 22. In other words, the relative fitness model assumes that performance scales between Bill and Sally for new workloads, as it did for the training data. In contrast, the absolute model assumes that Sally's performance will be similar to her past exams.

Applying the above analogy, because many workloads share relative fitness values, there could be fewer performance ratios to learn than performance values. As will be shown in Chapter 7, this is often the case. When using the same amount of training data, relative fitness models exceed the prediction accuracy of absolute models. Alternatively, relative fitness models can match the prediction accuracy of absolute models, while using less training data.

6.3 Model training

To train a relative fitness model, one needs to compare the performance of two storage systems over a variety of workload samples. Samples can be obtained from applications or, as discussed in Chapter 3,

Sample	Predictor variables	Predicted variable
1	$\mathbf{WC}_{j,1}$ $\mathbf{Perf}_{j,1}$ $\mathbf{Util}_{j,1}$	$P_{i,1}/P_{j,1}$
2	$\mathbf{WC}_{j,2}$ $\mathbf{Perf}_{j,2}$ $\mathbf{Util}_{j,2}$	$P_{i,2}/P_{j,2}$
n	$\mathbf{WC}_{j,n}$ $\mathbf{Perf}_{j,n}$ $\mathbf{Util}_{j,n}$	$P_{i,n}/P_{j,n}$

Table 6.1: Training data format for a relative fitness model. $\mathbf{WC}_j$, $\mathbf{Perf}_j$, and $\mathbf{Util}_j$ are vectors. P_i/P_j is a performance ratio (the relative fitness value).

Sample	Predictor variables	Predicted variable
1	$\mathbf{WC}_{j,1}$ $\mathbf{Perf}_{j,1}$ $\mathbf{Util}_{j,1}$	$P_{i,1}$
2	$\mathbf{WC}_{j,2}$ $\mathbf{Perf}_{j,2}$ $\mathbf{Util}_{j,2}$	$P_{i,2}$
n	$\mathbf{WC}_{j,n}$ $\mathbf{Perf}_{j,n}$ $\mathbf{Util}_{j,n}$	$P_{i,n}$

Table 6.2: Training data format for a relative performance model. $\mathbf{WC}_j$, $\mathbf{Perf}_j$, and $\mathbf{Util}_j$ are vectors. P_i is a performance metric.

I/O benchmarks, synthetic workload generators, or trace replay. The challenge is achieving adequate coverage. One must obtain samples that are representative of future workloads, but determining this *a priori* can be difficult. Moreover, even slight changes in the workload characteristics can translate to large differences in performance (e.g., adding a little spatial randomness to a workload can change throughput dramatically). In general, the best one can do is to make the most efficient use of the data that is available. As such, models that require fewer training samples are preferred. Likewise, models that explain the data in the least complex manner (Occam's razor) are preferred to models that potentially over-fit the data by confusing coincidental relationships for fact.

A workload sample is composed of three vectors: workload characteristics (**WC**), performance (**Perf**), and resource utilization (**Util**). Given multiple samples from two storage systems i and j (the same set of samples), the goal of a relative fitness model is to discover relationships between the predictor variables ($\mathbf{WC}_j$, $\mathbf{Perf}_j$, and $\mathbf{Util}_j$) and the predicted variable P_i (one of the performance metrics in $\mathbf{Perf}_i$). Table 6.1 shows the format of the training data for a relative fitness model. The formats for the relative performance model (Table 6.2) and the relative model (Table 6.3) are similar, but reflect Equations 6.2 and 6.1, respectively.

In contrast, an absolute model is only trained with workload samples from a single storage system. The goal of the absolute model is to discover relationships between the predictor variables ($\mathbf{WC}_i$) and the predicted variable P_i; the other metrics in $\mathbf{Perf}_i$ and $\mathbf{Util}_i$ are discarded. Table 6.4 illustrates.

Given training data like that shown in Tables 6.1 through 6.4, one can construct a black-box model using any number of learning algorithms. The problem falls under the general scope of *supervised learning*, where one has access to a set of predictor variables, as well as the desired response (the predicted variable) [4].

Sample	Predictor variables	Predicted variable
1	$\mathbf{WC}_{j,1}$	$\mathrm{P}_{i,1}$
2	$\mathbf{WC}_{j,2}$	$\mathrm{P}_{i,2}$
n	$\mathbf{WC}_{j,n}$	$\mathrm{P}_{i,n}$

Table 6.3: Training data format for a relative model. $\mathbf{WC}_j$ is a vector. P_i is a performance metric.

Sample	Predictor variables	Predicted variable
1	$\mathbf{WC}_{i,1}$	$\mathrm{P}_{i,1}$
2	$\mathbf{WC}_{i,2}$	$\mathrm{P}_{i,2}$
n	$\mathbf{WC}_{i,n}$	$\mathrm{P}_{i,n}$

Table 6.4: Training data format for an absolute model. $\mathbf{WC}_i$ is a vector. P_i is a performance metric.

6.4 Model selection (CART)

Relative fitness modeling is ideally suited to regression, where one wants to predict a continuous variable (i.e., a relative fitness value). There are many regression models in the literature [4]. This work focuses on the use of classification and regression trees (CART) [1] because of their simplicity, flexibility, and interpretability. As discussed in Chapter 5, CART models have been used successfully in the past to build absolute models [6].

6.4.1 A CART primer

Trees are built top-down, recursively, beginning with a root node. At each step in the recursion, the CART model-building algorithm determines which predictor variable in the training data best *splits* the current node into leaf nodes. The "best" split is that which minimizes the difference (e.g., root mean squared error, or RMS) among the samples in the leaf nodes, where the error for each sample is the difference between it and the average of all samples in the leaf node. In other words, a good split produces leaf nodes with samples that contain similar values. For example, consider the four training samples in Table 6.5. There are two predictor variables (request size and queue depth) and one predicted variable (the relative fitness of storage system i). For the first split, there are two options: the queue depth and the request size. As shown in Figure 6.1 (top half), splitting on the request size produces more homogeneous leaf nodes, and it is therefore the best first split. The CART algorithm then continues recursively on each subtree. The same predictor variable may appear at different levels in the tree, as it may be best to split multiple times on that variable. In this case, the queue depth is the next best split, resulting in the final tree shown in Figure 6.1 (bottom half).

Intuitively, the CART algorithm determines which of the predictor variables have the most information with respect to the predicted variable. The most relevant questions (splits) are asked first, and subsequent splits are used to refine the search. Trees are grown until no further splits are possible, thereby creating a *maximal tree*. A maximal tree is then *pruned* to eliminate over-fitting, and multiple pruning depths are explored. The optimal pruning depth for each branch is determined through *cross-validation* [5], in which a small amount of training data is reserved and used to test the accuracy of the pruned trees.

Request size	Queue depth	$RF_{j \to i}$
1 KB	1	.51
2 KB	2	.52
4 KB	1	.75
8 KB	2	.76

Table 6.5: An example of training samples that might be used to train a CART model.

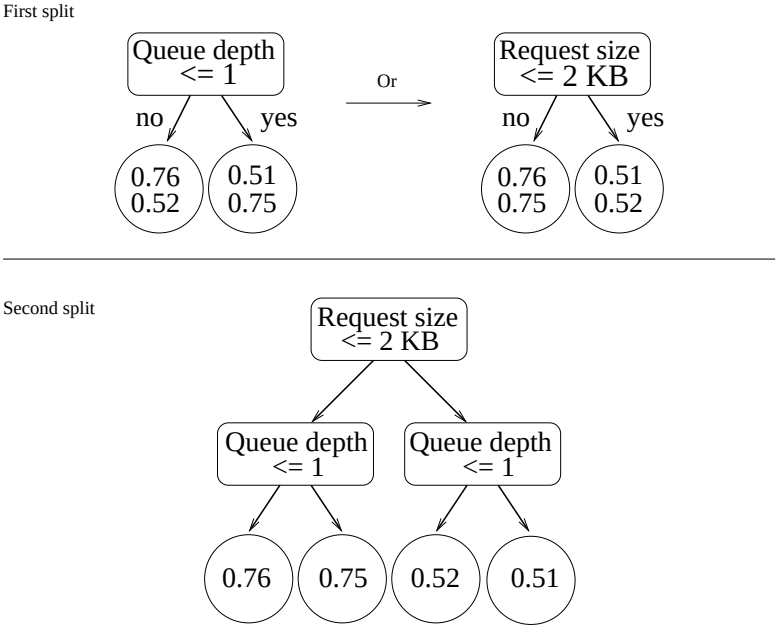

Figure 6.1: CART determines which split is best by inspecting the similarity of the samples in the resulting leaf nodes. Given the data in Table 6.5, splitting on the request size (right half) is better than splitting on the queue depth (left half), as it produces more homogeneous leaf nodes. For the second split, CART selects the queue depth.

Finally, to make a prediction for a new sample, one follows a path from the root node to a leaf node. As each node is visited, either the left or right branch is taken, depending on the splitting criterion and the values of the predictor variables for the sample. Leaf nodes contain the predicted variable that will be returned (e.g., the average or median value of all samples in the leaf node).

As can be seen in Figure 6.1, CART models are easy to read and interpret. As such, one can determine why a certain relative fitness value is being predicted by inspecting the internal nodes from the root to a leaf. A variety of real CART models will be explored in the next chapter.

6.5 Summary and concluding remarks

In summary, whereas conventional modeling constructs one absolute model per storage system and assumes that workload characteristics are unchanging across storage systems, the relative modeling ap-

proach construct two models for each pair of storage systems (*A to B* and *B to A*) and implicitly models the changing workloads. This approach also allows the use performance and resource utilization when making predictions, thereby reducing the dependency on workload characteristics.

A perceived cost of the relative approach is the additional model construction: $O(n^2)$ versus $O(n)$, where n is the number of storage systems. However, constructing a model takes at most a few seconds. Also, in the context of a data center, a storage system is usually a disk array or some other type of storage system with management processors that could be exploited for model construction. That is, each storage system could construct $n-1$ models that predict its performance relative to all other storage systems, so the computational resources could be made to scale. Also, in large-scale environments, certain collections of storage systems will be identical and can share models. In short, the cost of creating additional models is unlikely to be a significant issue.

The real cost of relative fitness modeling — a cost also shared by any statistical, absolute model — is the time required to obtain training data. However, relative fitness models have the potential for reducing this time, because they make more efficient use of the training samples by predicting performance ratios.

Bibliography

[1] Leo Breiman, Jerome Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Chapman and Hall, New York, NY, 1984.

[2] Douglas J. Futuyma. *Evolutionary Biology. Third Edition*. Sinauer Associates, December 1997.

[3] Johannes Gehrke. Decision Tree Construction. EBI Spring 2001.

[4] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer–Verlag. 2001, 2001.

[5] Tom M. Mitchell. *Machine Learning*. Series in Computer Science. McGraw-Hill, 1997.

[6] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R. Ganger. Storage device performance prediction with CART models. In *Proceedings of the 12th International Symposium on Modeling Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS-2004)*, Volendam, The Netherlands, October 2004. IEEE.

Chapter 7

Evaluation

This evaluation compares the prediction accuracy of absolute, relative, relative performance, and relative fitness models, using a variety of storage systems and application workloads. The storage systems are all disk arrays and, as described in Section 2.1, it is assumed that there is one workload per disk array. Appendix B contains an early follow-on to this work that predicts the performance impact when the disk arrays are being shared.

The disk arrays have different hardware platforms, software stacks, RAID configurations, number and type of disks, and cache sizes. The applications include Postmark [1], Cello trace replay [10], TPC-C [7], and a synthetic workload generator built for this evaluation, called Fitness [4]. Fitness is used to generate four different workload types: `FitnessDirect` (where I/O is issued directly to the arrays, with no host caching), `FitnessBuffered` (I/O is directed through the host buffer cache), `FitnessFS` (I/O is directed through a file in the host file system), and `FitnessCache` (the same as FitnessDirect, but with a varying working set size so as to explicitly test the array caches).

Five research hypotheses are evaluated via five experiments. Experiment 1 measures the change in block-level workload characteristics as the application workloads are moved across disk arrays. Experiment 2 evaluates absolute models, particularly how their prediction error increases as a result of changing workloads. Experiment 3 evaluates relative models and confirms that the pairwise modeling of disk arrays can help reduce the effects of changing workloads. Experiment 4 evaluates relative performance models and confirms that one can use the performance of array j to predict performance of array i. Experiment 5 evaluates relative fitness models and confirms that performance ratios (relative fitness values) further improve prediction accuracy. In Experiments 2–5, the metric chosen for comparing prediction accuracy is the relative error:

$$relative\ error = \frac{predicted\ value - measured\ value}{measured\ value}$$

7.1 Executive summary

Figure 7.1 summarizes the results of Experiments 2–5. It plots the cumulative distribution of relative error for each of the models evaluated. The CDFs are constructed using all of the predictions from the

evaluation, of which there are over 5000 per model. The different error distributions visually confirm each hypothesis.

First, the prediction error of an absolute model, in practice, increases due to changing workload characteristics. Specifically, when workload characteristics are obtained from one array j and input into the model of a different array i, there is more error (lowest CDF line in each graph) than the idealized case in which characteristics are obtained from the same array for which a prediction is being made.

Second, a relative model can reduce this error. As can be seen in the figure, particularly the latency graph, the relative models have less error than the in-practice absolute models (i.e., where $i \neq j$). In fact, one can see that the error distributions of the idealized absolute models (where $i = j$) and the relative models (where $i \neq j$) are nearly identical. In other words, a relative model has similar error to an idealized absolute model, thereby confirming that relative modeling can remove or eliminate the prediction errors due to changes in the workload characteristics.

Third, a relative performance model reduces error further, confirming that the observed performance of array j can be used to predict the performance of a different array i. As can be seen in Figure 7.1, the relative performance models improve upon the relative models and absolute models.

Finally, as shown by the error distributions of the relative fitness models (the top-most CDF line in each graph), performance ratios are better predictors than performance values. In each graph, the relative fitness models produce the least error.

Overall, when comparing the relative fitness models to the absolute models, one can see significant improvements in prediction accuracy. The average median relative error of the bandwidth predictions (i.e., the median relative bandwidth prediction error for each application is computed, and these medians are then averaged) is reduced from 23% to 11%, throughput from 17% to 10%, and latency from 21% to 14%. Also, the relative fitness models reduce prediction error with simpler models (fewer rules). On average, the relative fitness models in this evaluation are 22% smaller than the absolute models (i.e., relative fitness models have 22% fewer leaf nodes in the CART models).

Moreover, the relative fitness models do not require as much training data. Figure 7.2 plots the average median relative prediction error of the applications, for various training/testing splits. The size of the training set varies from 25% of the collected application workload samples to 75%, in increments of 5%. For each training set size, models are trained using x% of the data and tested using the remainder. As such, Figure 7.2 illustrates how the models "learn" as more workload samples are used for training. As can be seen in the Figure, the absolute models (AM) produce the most error, the relative models (RM) reduce the error caused by changing workload characteristics, the relative performance models (RM') reduce error further, and the relative fitness models (RF) produce the least error. In all cases, the relative fitness models are more accurate than the absolute models, even when trained with significantly fewer samples.

7.2 Organization

This evaluation is organized as follows. Section 7.3 describes the experimental methodology, including the research hypotheses, the data collection, and the set of experiments. Section 7.4 describes the experimental setup, including the application servers and the storage arrays, the application workloads, the

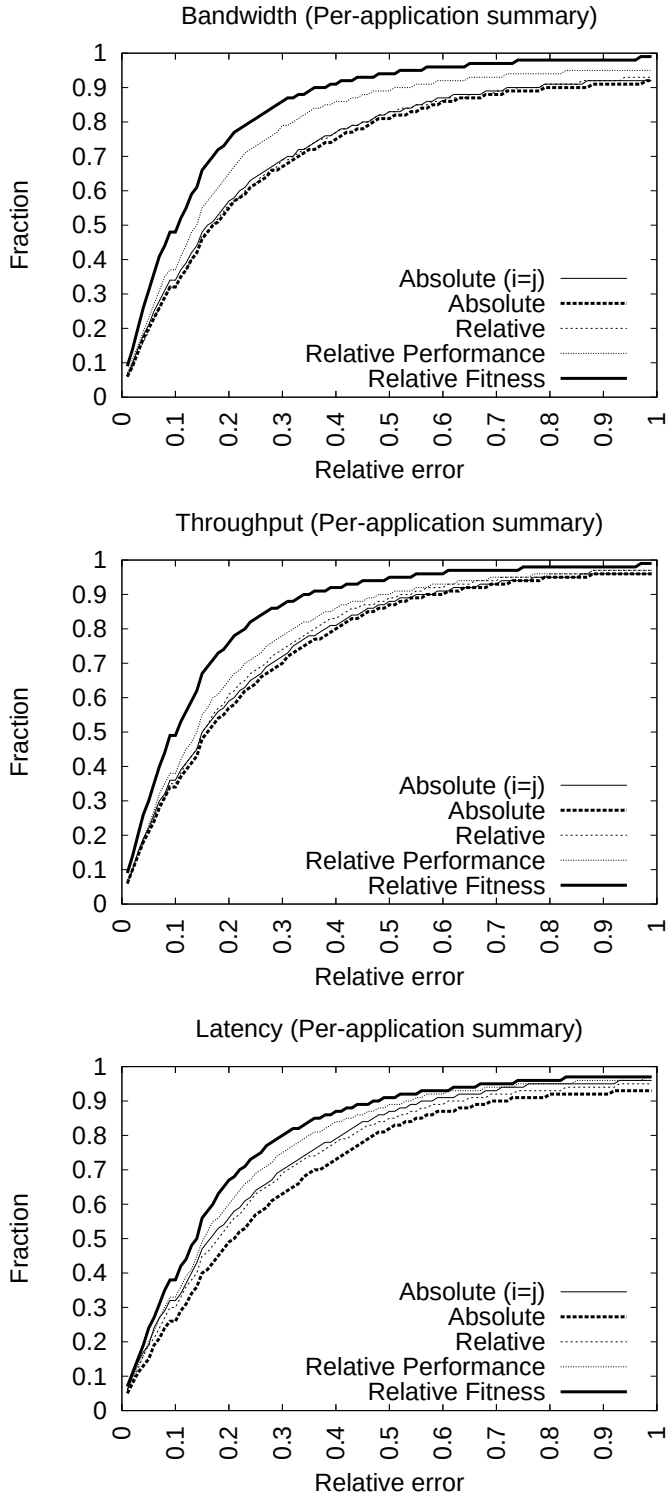

Figure 7.1: The cumulative distributions of the relative prediction error for the idealized absolute model ($i = j$), the in-practice absolute model ($i \neq j$), the relative model, the relative performance model, and the relative fitness model. The thick solid line (least error) is that of the relative fitness models. The thick dashed line (most error) is that of the in-practice absolute models.

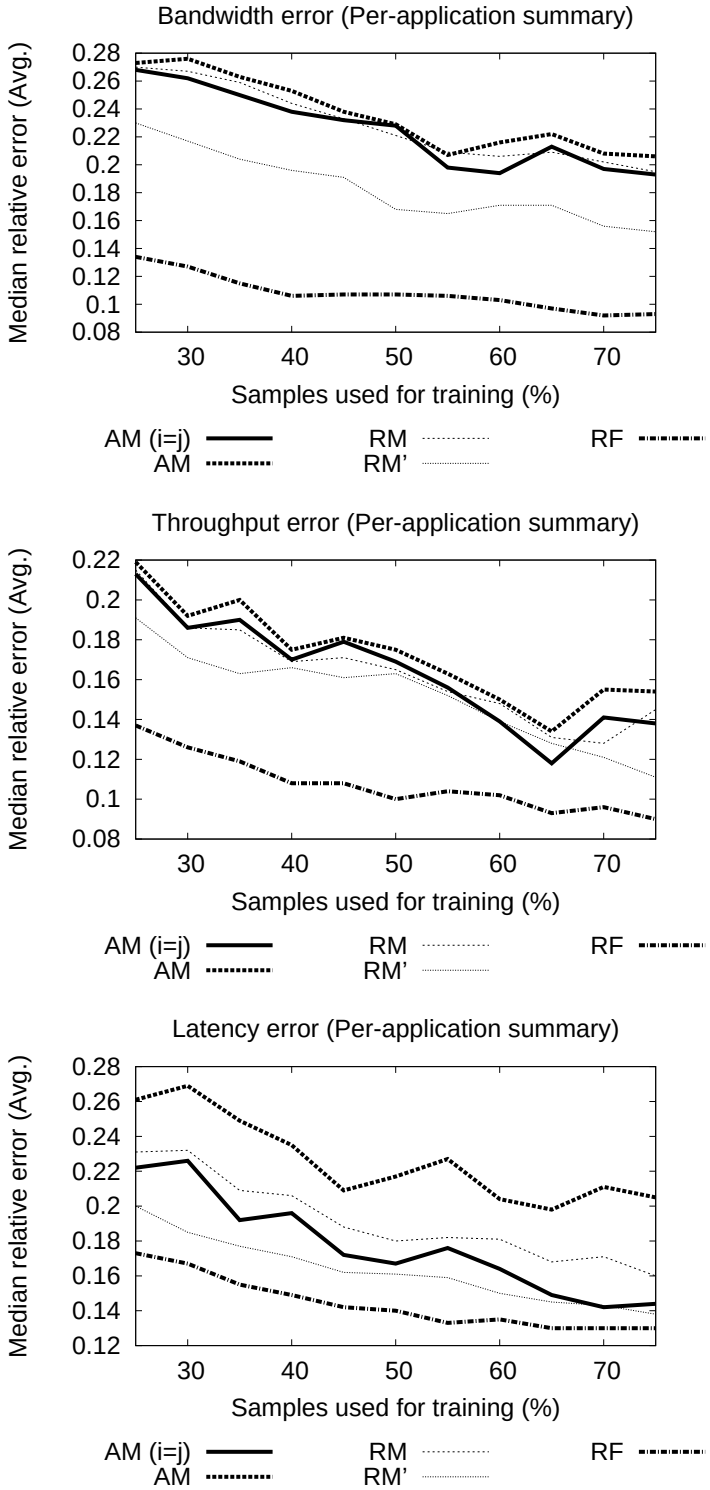

Figure 7.2: Error vs. training set size. The training set size varies from 25% of the collected samples to 75%; the remaining samples are used for testing. For each training set size, an error metric (median relative error) is calculated for each pairing of arrays. These metrics are then averaged across all pairings to produce the values shown.

workload characteristics and performance metrics, and the machine learning infrastructure. Sections 7.5 through 7.9 present the experimental results. All data is contained in the appendices, but representative tables and graphs are copied into the evaluation sections as necessary. One need only refer to an appendix if additional detail is desired.

For each experiment, the results from each application are presented in turn. Particular attention is given to the first application (`FitnessDirect`), so as to provide instruction on interpreting the tables and figures. Only high-level performance summaries and anecdotes are presented for the remaining applications.

7.3 Experimental methodology

A collection of application workloads are run on various disk arrays in order to characterize their I/O and measure their performance. As described in Chapters 3 and 4, both the performance and workload characteristics of each application are measured at the block-level, below all file systems and OS caches. The collected samples are then used in five different experiments. Each experiment is designed to test one of the research hypotheses.

7.3.1 Research hypotheses

Summarizing from the preceding chapters, there are five research hypotheses:

Hypothesis 1 (Changing workloads): When storage arrays j and i have different performance characteristics, the block-level workload characteristics of an application can change as the application is moved from array j to array i (i.e., $\mathbf{WC}_i \neq \mathbf{WC}_j$).

Hypothesis 2 (Absolute models): The change in workload characteristics is significant enough to affect the accuracy of an absolute model of array i (Equation 5.3) when the workload characteristics are obtained from a different array j. More specifically, absolute models are most accurate when the workload characteristics are obtained from the same array for which the performance prediction is being made. However, as discussed, this ideal situation will not occur in practice.

Hypothesis 3 (Relative models): Because relative models (Equation 6.1) are trained to predict the performance of array i relative to the workload characteristics of array j, they are less sensitive to the changes in the workload characteristics and are, therefore, more accurate than absolute models.

Hypothesis 4 (Relative performance models): Unlike an absolute model, a relative model can distinguish workloads with different performance and similar, if not identical, workload characteristics. This is trivially accomplished by adding performance information to the model (Equation 6.2). A relative model trained with performance information is, therefore, more accurate than a relative model trained only with workload characteristics.

Hypothesis 5 (Relative fitness models): Relative fitness models (Equation 6.3) are more accurate than relative performance models, as performance ratios naturally interpolate among known training

samples. Stated differently, linear functions (performance ratios) are better predictors than constant performance values.

7.3.2 Data collection

As described in Chapter 3, each application is used to generate multiple workload samples. A sample contains the observed workload characteristics and performance over a specified testing period, after the application has been allowed to run for a specified warming period. The warming and testing periods are chosen separately for each application, such that performance reaches a steady state (e.g., the caches have been sufficiently warmed and performances "levels off"). Multiple samples are obtained from each application by assigning random values to the application parameters and running the application multiple times. The same set of samples are collected from each storage array (i.e., the random number generators are seeded identically).

Each workload sample is run for three iterations. The best performing iteration of each sample is retained for model training and testing, and it is also used to measure the variance in performance and workload characteristics across disk arrays, referred to as the *inter-array variance*. This is done separately for each metric. For example, when reporting and predicting the average bandwidth, the iterations with the best bandwidth are used. The remaining two iterations are used to measure the variance of a sample on a given array, or the *intra-array variance*.

7.3.3 Description of experiments

Each of the following experiments is designed to test one of the research hypothesis:

Experiment 1 (Changing Workloads): Experiment 1 quantifies performance differences across the disk arrays and the extent to which the workload characteristics change because of these differences. For each application, the intra-array and inter-array variance (both workload characteristics and performance) of the samples are calculated. There are two objectives. The first objective is to show that the intra-array variance is low (i.e., that identical samples, run on the same disk array, will have close to identical performance and workload characteristics). The second objective is to show that the inter-array variance is high (i.e., that identical samples, run on different arrays, can have different performance and workload characteristics).

Experiment 2 (Absolute models): Experiment 2 measures the effect that changing workloads have on the prediction accuracy of an absolute model (Equation 5.3). Models are trained for each combination of disk array and performance metric; a portion of the sample workloads are used for model training, and the rest are used for model testing. The models are evaluated across all pairings of disk arrays.

Experiment 3 (Relative models): Experiment 3 compares the accuracy of a relative model (Equation 6.1) to that of an absolute model. Models are evaluated for each combination of array pairing and performance metric. As per Equation 6.1, only workload characteristics are used to train the models.

Experiment 4 (Relative performance models): Experiment 4 compares the accuracy of a relative performance model (Equation 6.2) to that of a relative model. Relative performance models are evaluated for each combination of array pairing and performance metric. Unlike a relative model, which only uses workload characteristics, a relative performance model is also trained with performance observations.

Experiment 5 (Relative fitness models): Experiment 5 compares the accuracy of a relative fitness model (Equation 6.3) to that of a relative performance model. Relative fitness models are evaluated for each combination of array pairing and performance metric. Unlike the absolute, relative, and relative performance models, the relative fitness models predict performance ratios (relative fitness values) rather than performance values.

7.4 Experimental setup

7.4.1 Initiator and target platforms

All experiments are run on IBM x345 servers (dual 2.66 GHz Xeon, 1.5 GB RAM, GbE, Linux 2.6.12), directly attached to one of four Internet SCSI (iSCSI) [13] disk arrays using copper gigabit Ethernet. The direct connection eliminates any potential switch congestion within the shared lab infrastructure. The servers communicate with each array using an open source iSCSI device driver [4]. The device driver contains counters, below the file system and page cache, that are used to characterize workloads and measure their performance.

The disk arrays have different hardware and software platforms, and different configurations:

Vendor 1 is a 4-disk IBM x345 server running Intel's open source iSCSI target stack [4]. The disk array is configured as RAID-0 (striping) using Linux software RAID. The disk drives are 36 GB 10K RPM IBM SCSI drives and are accessed in raw mode (i.e., no caching in the RAID controller).

Vendor 2 is a 6-disk Intel Mt. Jefferson storage system (SSR313MJ2) [5] with a LeftHand iSCSI storage stack [11]. The array is configured as RAID-0 and has 512 MB of cache. The disk drives are 250 GB 7200 RPM Seagate Barracuda SATA drives.

Vendor 3 is an 8-disk Intel Mt. Adams storage system (SSR212MA) [6] with an Open-E iSCSI storage stack [12]. The array is configured as RAID-10 (striping over 4 mirrored pairs) with 512 MB of cache. The disk drives are 250 GB 7200 RPM Seagate Barracuda SATA drives.

Vendor 4 is a 14-disk EqualLogic PS100E storage array [8]. The array is configured as RAID-50 (striping over two 7-disk RAID-5 arrays) and has 1 GB of cache. The disk drives are 400 GB 7200 RPM Hitachi Deskstar SATA drives.

The arrays are anonymized for this evaluation. Each is referred to, in some random order, as Array A, Array B, Array C, and Array D.

7.4.2 Application workloads

Workload samples are collected from four applications: Fitness [4] (synthetic I/O), Postmark [1], Cello [10] trace replay, and TPC-C [7]. Fitness is used to automate the sample collection process. That is, in addition to generating synthetic I/O, Fitness is a test harness for characterizing the I/O and measuring the performance of other applications. The workload characteristics and performance are obtained from the iSCSI device driver [4] through the `/proc` file system. Details on the workload characterization process are presented in Section 7.4.3.

For each sample, Fitness optionally formats and mounts a 16 GB Ext2 file system, starts the application (or the internal workload generator), warms for specified period, tests for a specified period (30 seconds for all applications in the evaluation), records the workload characteristics and performance over the test window, exits the application, optionally unmounts the file system, and then flushes the storage array's cache. The flush is accomplished by reading a large sequential region of each array, twice.

Each application is described in more detail below. Appendix A contains the actual Fitness command-line arguments and scripts used to generate the workload samples.

Fitness When run as a synthetic workload generator, the arguments to Fitness include the byte range (or file size), write percent, write request size (KB), read request size (KB), number of outstanding requests (i.e., queue depth, level of concurrency, or multi-programming level), write randomness, and read randomness. The workload parameters are randomly selected for each sample. Specifically, the write percent takes on a value from 0 to 100 (stepping 20), the write and read request sizes each take on a value from 1 to 256 KB (stepping 1 KB), the write and read randomness a value from 0 to 100 (stepping 20), and the queue depth a value from 1 to 64 (stepping powers of 2). For the randomness parameters, a value of x means that x% of the I/Os are uniformly random and $100 - x$ are sequential. All requests are aligned according to the chosen request sizes (e.g., if a 1 MB file is accessed with a write request size of 1 KB, then there are 1024 possible offsets for performing the write I/O).

Fitness is configured in one of three I/O modes: *direct* mode to a block device (i.e., using the `O_DIRECT` flag during `open()`), *buffered* mode to a block device (i.e., though the OS buffer cache), and *file* mode through a formatted Ext2 file system. Each mode is considered a separate workload in this evaluation. Respectively, the workloads are referred to as `FitnessDirect`, `FitnessBuffered`, and `FitnessFS`. `FitnessDirect` and `FitnessBuffered` perform I/O to a 16 GB region of the disk (no file system), and `FitnessFS` creates a 4 GB file. All three modes perform I/O to the region/file according to the randomly selected workload parameters. 200 samples are generated for each mode.

A fourth run of Fitness is used to test workloads that hit completely-in or completely-out of the array caches. This workload, called `FitnessCache`, is a random, read-only workload. The read request size varies from 1 to 64 KB (powers of 2), and the queue depth varies from 1 to 16 (powers of 2). The byte range is either 16 MB (completely in all caches) or 16 GB (completely outside of the caches). Only 70 workloads are possible (7 request sizes $\times$ 5 queue depths $\times$ 2 ranges). So, rather than randomly sample the workload space, each workload is specified in turn.

In total, 670 workload samples are collected from each storage array: 200 for `FitnessDirect`, 200

for `FitnessBuffered`, 200 for `FitnessFS`, and 70 for `FitnessCache`. Again, each sample is run for three iterations, and the best performing iterations are retained for model training and testing.

Postmark The Postmark benchmark [1] simulates small-file Internet traffic (news, e-mail, and WWW). In the first phase of Postmark, a large file pool is created across multiple subdirectories. In the second phase, transactions are performed against the file pool, including file creation, deletion, writing, appending, and reading.

50 runs of Postmark are generated. For each run, the parameters are randomly selected. The file size varies from 1 B to 1 MB (stepping 1) and the number of subdirectories from 1 to 1000 (stepping 1). The number of files is selected so as to create an initial pool capacity of approximately 2 GB. In other words, the number of files is set to 2 GB divided by the randomly selected file size.

Two workload samples are obtained from each run: one from the file creation phase and a second from the transactions phase. This yields a total of 100 samples from each disk array.

Cello99 The Cello traces are a collection of block-level I/O traces from a file server at HP laboratories [10]. The traces used are from a two-week period: 11/08/99 through 11/12/99 and 11/15/99 through 11/19/99. Each trace covers a 24-hour period. The 10 day-long traces are converted from binary to ASCII and separated into 80 3-hour traces. Each is considered a separate workload sample, and Fitness is used to perform the replay (using the `--replay` option).

All 80 traces are replayed through the OS buffer cache using an as-fast-as-possible arrival process (no think time between each I/O). Although the traces at HP Labs were collected below the page cache (at the SCSI device driver level), in this evaluation they are replayed above the caches in order to introduce caching effects that normally arise in a system. Without information as to how the I/O would change (e.g., due to feedback effects between a file system and storage system), this is perhaps the best one can do. Similarly, because the true multi-programming level is unknown, a random level is selected by the Fitness replayer for each sample (1,2,4,8,16, 32, or 64).

Unlike the other workloads, which have steady-state I/O performance after a certain amount of warming, the Cello workloads are variable. As such, replay may progress at different rates on each array, and therefore be at different locations in the trace after a specified warming period. Because of this, the Cello replay warms until 128 MB of data have been transferred (using the `--warm_mb` option in `Fitness`), rather than for a fixed amount of warming time. 128 MB is chosen, as opposed to a larger warming period, to avoid running out of trace records before a performance measurement can be taken (i.e., if too much time is spent warming, most of the Cello traces return EOF during replay).

TPC-C [7] TPC Benchmark C is an online transaction processing (OLTP) benchmark from the Transaction Processing Performance Council. It simulates a computing environment where users submit transactions against a database, including order placement, order status, payment processing, and inventory checks. In this work, TPC-C is run on the Shore storage manager [3], configured with 8 KB pages, 10 warehouses, and 10 clients; the storage footprint is 5 GB. 50 runs of TPC-C are generated, each with random probabilities selected for `prob_neworder`, `prob_payment`, `prob_order_status`, `prob_delivery` and `prob_stock_level` [7], which must sum to 1.

Counter	Meaning	Counter	Meaning
ops	Total operations	op_latencies_wr	Sum of write latencies
ops_wr	Write operations	op_latencies_rd	Sum of read latencies
ops_rd	Read operations	op_depths_wr	Sum of write queue depths
bytes_sent	Bytes written	op_depths_rd	Sum of read queue depths
bytes_recv	Bytes read	op_jumps_wr	Sum of write jump distances
op_latencies	Sum of latencies	op_jumps_rd	Sum of read jump distances

Table 7.1: Counters maintained by the iSCSI device driver.

7.4.3 Workload characteristics and performance metrics

Workload characteristics and performance metrics are calculated from kernel counters in the iSCSI device driver, available to a user application (Fitness in this case) through the `/proc` file system. The counters are updated by the driver at the completion of every I/O, including operation counts, bytes transferred, operation latencies, queue depths (an approximation of the true multi-programming level of an application), and I/O jump distances (one measure of the spatial randomness). Note, the latency, queue depth, and jump distance counters are running sums. When an I/O completes, its latency, queue depth and jump distance are added, respectively, to each of these counters. This enables one to calculate, say, the average write queue depth — by dividing the sum of write queue depths by the total number of write requests. Also, the iSCSI driver maintains separate "file pointers" for writes and reads, which are simply the offsets of the last write and read request issued to a disk array. These file pointers are used in the device driver to calculate the jump distance between successive writes and reads to an array.

For each workload sample, Fitness queries the counters once after the warming period and once after the testing period. Then, the difference between the two sets of counters is used to calculate averages for the workload characteristics and performance. Table 7.1 summarizes the counters used by Fitness.

The workload characteristics output by Fitness (from the driver counters) are similar to its input parameters. They include, as described in Chapter 4, the average write fraction (0 to 1), write request size (KB), read request size, write randomness (jump distance before each write, expressed in MB), read randomness, write queue depth (number of outstanding writes), and read queue depth. The performance metrics, as described in Chapter 3, include the average bandwidth (MB/sec), throughput (IO/sec), request latency (ms), write request latency, and read request latency. Table 7.2 summarizes each workload characteristic and performance metric. The equations below contain the actual calculations.

Read/write ratio

The write fraction (`wr`) is the fraction of I/Os that are write requests. Because $\texttt{ops\_wr}_{warm}$ represents the number of write operations seen by the device driver after the warming period, and $\texttt{ops\_wr}_{test}$ represents that after the testing period, the total number of write operations is $\texttt{ops\_wr}_{test} - \texttt{ops\_wr}_{warm}$. Similarly, $\texttt{ops}_{test} - \texttt{ops}_{warm}$ represents the total number of I/Os (write or read). Therefore, the fraction of I/Os that are writes is calculated as follows:

$$\texttt{write fraction} = \frac{\texttt{ops\_wr}_{test} - \texttt{ops\_wr}_{warm}}{\texttt{ops}_{test} - \texttt{ops}_{warm}}$$

Workload characteristic	Units	Variable
Write fraction	0 to 1	`wr`
Write request size	KB	`wrsz`
Read request size	KB	`rdsz`
Write jump distance	MB/IO	`jmp_wr`
Read jump distance	MB/IO	`jmp_rd`
Write queue depth	IOs	`qdep_wr`
Read queue depth	IOs	`qdep_rd`
Performance measurement	**Units**	**Variable**
Bandwidth	MB/sec	`bw`
Throughput	IO/sec	`iops`
Latency	ms/IO	`lat`
Write latency	ms/IO	`lat_wr`
Read latency	ms/IO	`lat_rd`

Table 7.2: Workload characteristics and performance metrics output by Fitness.

Request sizes

The write request size (`wrsz`) and read request size (`rdsz`) are the average I/O request sizes (KB):

$$\texttt{write request size} = \frac{(\texttt{bytes\_sent}_{test} - \texttt{bytes\_sent}_{warm})/1024}{\texttt{ops\_wr}_{test} - \texttt{ops\_wr}_{warm}}$$

$$\texttt{read request size} = \frac{(\texttt{bytes\_recv}_{test} - \texttt{bytes\_recv}_{warm})/1024}{\texttt{ops\_rd}_{test} - \texttt{ops\_rd}_{warm}}$$

Queue depths

The write queue depth (`qdep_wr`) and read queue depth (`qdep_rd`) are the average number of outstanding requests when a new I/O is issued:

$$\texttt{write queue depth} = \frac{\texttt{op\_depths\_wr}_{test} - \texttt{op\_depths\_wr}_{warm}}{\texttt{ops\_wr}_{test} - \texttt{ops\_wr}_{warm}}$$

$$\texttt{read queue depth} = \frac{\texttt{op\_depths\_rd}_{test} - \texttt{op\_depths\_rd}_{warm}}{\texttt{ops\_rd}_{test} - \texttt{ops\_rd}_{warm}}$$

Spatial randomness

The write randomness (`jmp_wr`) and read randomness depth (`jmp_rd`) are, respectively, the average jump distance for each write (relative to the last write offset) and each read (relative to the last read offset):

$$\texttt{write randomness} = \frac{\texttt{op\_jumps\_wr}_{test} - \texttt{op\_jumps\_wr}_{warm}}{\texttt{ops\_wr}_{test} - \texttt{ops\_wr}_{warm}}$$

$$\texttt{read randomness} = \frac{\texttt{op\_jumps\_rd}_{test} - \texttt{op\_jumps\_rd}_{warm}}{\texttt{ops\_rd}_{test} - \texttt{ops\_rd}_{warm}}$$

Bandwidth

Bandwidth (`bw`) is the average data transfer rate of the application over the specified testing period. Fitness records the wall clock time after the warming period ($\texttt{secs}_{warm}$) and after the testing period ($\texttt{secs}_{warm}$). Therefore, bandwidth is calculated as follows:

$$\texttt{bandwidth} = \frac{(\texttt{bytes\_sent}_{test} - \texttt{bytes\_sent}_{warm}) + (\texttt{bytes\_recv}_{test} - \texttt{bytes\_recv}_{warm})}{\texttt{secs}_{test} - \texttt{secs}_{warm}}$$

Throughput

Throughput (`iops`) is the average I/O rate:

$$\texttt{throughput} = \frac{\texttt{ops}_{test} - \texttt{ops}_{warm}}{\texttt{secs}_{test} - \texttt{secs}_{warm}}$$

Latency

Latency (`lat`) is the average I/O request latency (i.e., *driver completion time* − *driver issue time*). It can also be calculated separately for writes (`lat_wr`) and reads (`lat_rd`):

$$\begin{aligned}
\texttt{latency} &= \frac{\texttt{op_latencies}_{test} - \texttt{op_latencies}_{warm}}{\texttt{ops}_{test} - \texttt{ops}_{warm}} \\
\texttt{write latency} &= \frac{\texttt{op_latencies_wr}_{test} - \texttt{op_latencies_wr}_{warm}}{\texttt{ops_wr}_{test} - \texttt{ops_wr}_{warm}} \\
\texttt{read latency} &= \frac{\texttt{op_latencies_rd}_{test} - \texttt{op_latencies_rd}_{warm}}{\texttt{ops_rd}_{test} - \texttt{ops_wr}_{warm}}
\end{aligned}$$

7.4.4 Machine learning infrastructure

Salford Systems' CART is used for the evaluation [14]. This particular implementation of CART is based on the original code and direct collaboration with the UC Berkeley statisticians who invented CART in 1984 [2]. As described in Section 6.3, the input into Salford's CART algorithm is a training data file that contains a collection of workload samples. Each row in the file is a workload sample, and each column is an attribute (a workload characteristic or performance metric). Exactly one attribute is the target function, or dependent variable, that a CART regression model is trained to predict. The predictor variables are selected from the remaining attributes.

Attributes are selected by CART as described in Section 6.4. Salford's CART is specifically configured to split on attributes that produce leaf nodes with the *least absolute deviation* among the training samples. As described in Section 6.4, Salford's CART algorithm grows a maximal tree, in which no further splits are possible, and then prunes the tree to help reduce over-fitting. Various pruning depths are explored, resulting in multiple candidate trees, and not all branches in a tree are necessarily pruned to the same

depth. Among the candidate trees, Salford's CART selects the "optimal" tree, or that which produces the lowest 10-fold cross-validation error over the training samples.

Salford's CART provides a variety of useful data-mining information along with the tree model, including an analysis of the tree-building process (cross-validation, competitor splits, and pruning information) and a description of the resulting tree, including the number of terminal nodes (or leaves) and the splitting criteria for each internal node. The number of terminal nodes is used to describe the complexity of a tree and is referenced in the evaluation. As described in Section 6.3 (Occam's razor), trees with fewer terminal nodes are preferred, so as to prevent over-fitting.

From a given tree, this evaluation computes an *importance measure* for each attribute. The score is simply the total number of samples that are split by that attribute, over the entire tree, normalized by the attribute that split the most samples. As such, the score can be used to rank the attributes from "most information" to least. Using this scoring method, intuitively, the attributes involved in the most splits are the ones providing the most useful information. Alternatively, one could visually inspect each tree to see which attributes are being used by CART.

7.5 Experiment 1: changing workloads

Experiment 1 is presented in two parts. First, Section 7.5.1 presents a performance analysis of the application workloads and disk arrays. It establishes that the performance differences among the arrays are significant, yet also shows that arrays can react in similar ways to changes in workload characteristics. For example, increasing the queue depth or request size usually increases an array's performance. Second, Section 7.5.2 presents the workload characteristics of the application samples and quantifies how they change across arrays.

Because the training and testing samples are generated in the same manner, by running each application multiple times with uniformly random parameters, only the training samples are presented in each section. The testing samples have similar distributions of performance and workload characteristics.

7.5.1 Performance analysis

Disk arrays can vary in performance, yet still exhibit similar behavior.

This section summarizes the performance behavior of the disk arrays, for each of the application workloads described in Section 7.4.2. The performance (bandwidth, throughput, and latency) of each workload sample is calculated as per Section 7.4.3. Then, the samples are averaged for each application and presented in Table 7.3. Recall that each sample is run for three iterations and that the highest performing iteration of each sample is used in computing the averages. For example, suppose that only two samples were obtained from an application: the first sample had a performance measurement of 32 MB/sec for iteration 1, 33 MB/sec for iteration 2, and 31 MB/sec for iteration 3; similarly, a second (different) sample had performance measurements of 13 MB/sec, 10 MB/sec, and 12 MB/sec. Then, the average performance reported in Table 7.3 would be the average of the best performing iterations (33 MB/sec and 13 MB/sec), or 23 MB/sec.

As described in Chapter 3, three statistics are used to compare the averages in Table 7.3: their mean absolute deviation (MAD), their coefficient of variation (COV), and their maximum relative difference (Max. Diff.). The purpose of the comparison is to quantify the change in performance across arrays. More specifically, the MAD statistic is the mean deviation of the averages in Table 7.3, the COV statistic is the mean of the averages divided by their standard deviation (expresses as a percent), and Max. Diff. is simply the relative difference, or range, of performance between the fastest and slowest array (also expressed as a percent). Throughout this evaluation, only Max. Diff. is referenced in the text. The MAD and COV statistics are only provided for reference.

To ensure that the changes in performance are due primarily to the performance characteristics of the disk arrays, rather than variance in the measurements, the average value of each performance metric is calculated a second time, using the second-best iteration of each sample. Then, the relative difference in average performance between the best and second-best iterations, called "Rel. Diff.," is calculated. These values can be found in the Appendices. To continue with the previous example, the average performance of the second best iterations would be 22 MB/sec (i.e., the average of 12 MB/sec and 32 MB/sec). Therefore, the relative difference in performance between the best and second-best iterations is $\frac{23-22}{23} \times 100$, or 4.5%.

One can compare the Rel. Diff. value directly to the Max. Diff. presented in Table 7.3. If Max. Diff. is greater (which it is, in all cases), one can conclude that the changes in performance are not related to variance. This is admittedly a pedantic exercise for the performance metrics which vary by a considerable amount across arrays, clearly more than just variance. However, these same statistics are used when quantifying the changes in workload characteristics, which can be more subtle, and are therefore introduced here. Throughout this section, unless otherwise stated, one can assume that the average intra-array variance in performance for each sample (Rel. Diff.) is small (i.e., $< 5\%$).

Arrays are ranked from "fastest" to "slowest" based on the average bandwidths reported in Table 7.3 (throughput or latency could have also been used). In addition, a variety of distribution functions and performance graphs are used to illustrate how performance varies across the arrays. Specifically, a cumulative distribution function (CDF) of each performance metric is provided, composed of the measured performance of each of the training samples. These CDFs can be quickly eye-balled to see if arrays have similar performance distributions. Then, the training samples of each application are sorted, according to the performance of one of the arrays, and their performance is plotted. In other words, the samples of one array are sorted from lowest to highest in terms of their performance. Then, each of the other three arrays are sorted in the same order. Such sorted performance graphs allow one to easily see how the performance of the arrays differs on a sample by sample basis. Of course, such a graph is necessary given the multiple workload dimensions. Array C is chosen to establish the sort order, as it tends to be the slowest array and produce the most uncluttered graphs.

Finally, to supplement the performance graphs, a set of additional microbenchmarks (sequential writes, random writes, sequential reads, and random reads) are used to illustrate how the performance of each array changes as two workload characteristics are varied: request size and queue depth. These samples are a subset of the `FitnessDirect` workload samples, sorted by increasing queue depth and request size.

`FitnessDirect`

Beginning with `FitnessDirect`, as shown in Table 7.3, the average bandwidths of Arrays A, B, C, and D are 30 MB/sec, 30 MB/sec, 13 MB/sec, and 37 MB/sec, respectively. Again, these averages are computed using the best iterations from each of the 100 training samples of `FitnessDirect`. The maximum relative difference in average bandwidth (Max. Diff.) occurs between Array C (13 MB/sec) and Array D (37 MB/sec): $\frac{37.42-12.93}{12.93}$, or 189%. In other words, the average bandwidth of `FitnessDirect` varies by as much as 189% across disk arrays. Similarly, average throughput varies by up to 281% and average latency by 259%. On average, Array D is the fastest array, Array C is the slowest, and Arrays A and B are about equal.

To visualize these performance differences, the left half of Figure 7.3 plots the cumulative distribution function of each performance metric. For example, approximately 90% of the `FitnessDirect` samples have an average bandwidth of about 15 MB/sec or less on Array C. Also, a quick inspection of the CDFs indicate the arrays with similar performance distributions. In particular, note the similarity among Arrays A, B, and D; Array C is an outlier. Alternatively, probability distribution functions of each performance metric can be found in the Appendix for `FitnessDirect` training. Each PDF illustrates the range of performance values for the `FitnessDirect` samples; clusters of commonly seen performance values can be seen in the graphs.

FitnessDirect							
	ArrayA	**ArrayB**	**ArrayC**	**ArrayD**	**MAD**	**COV**	**Max. Diff.**
Bandwidth (MB/sec)	30.42	30.05	12.93	37.42	7.4	32.6%	189.4%
Throughput (IO/sec)	280.10	360.82	116.16	442.60	101.8	40.2%	281.0%
Latency (ms)	25.03	25.27	81.70	22.75	21.5	64.2%	259.1%
Write latency (ms)	25.14	11.40	77.46	14.54	22.7	83.0%	579.5%
Read latency (ms)	17.38	40.74	60.74	28.27	14.0	43.8%	249.5%
FitnessBuffered							
	ArrayA	**ArrayB**	**ArrayC**	**ArrayD**	**MAD**	**COV**	**Max. Diff.**
Bandwidth (MB/sec)	27.54	20.96	16.18	28.83	4.8	21.9%	78.2%
Throughput (IO/sec)	372.61	280.07	225.89	421.02	71.9	23.5%	86.4%
Latency (ms)	34.78	35.61	74.23	28.05	15.5	42.1%	164.6%
Write latency (ms)	66.61	52.38	146.08	36.67	35.3	55.9%	298.4%
Read latency (ms)	13.78	21.20	25.35	16.38	4.1	23.2%	84.0%
FitnessFS							
	ArrayA	**ArrayB**	**ArrayC**	**ArrayD**	**MAD**	**COV**	**Max. Diff.**
Bandwidth (MB/sec)	23.48	17.59	15.43	23.88	3.6	18.3%	54.8%
Throughput (IO/sec)	302.07	227.98	203.18	316.14	46.8	18.2%	55.6%
Latency (ms)	24.69	21.51	49.03	17.55	10.4	43.6%	179.4%
Write latency (ms)	51.24	39.88	116.70	23.68	29.4	61.1%	392.8%
Read latency (ms)	9.84	15.58	16.99	13.15	2.4	19.5%	72.7%
FitnessCache							
	ArrayA	**ArrayB**	**ArrayC**	**ArrayD**	**MAD**	**COV**	**Max. Diff.**
Bandwidth (MB/sec)	16.18	15.66	35.25	40.11	10.4	43.5%	156.1%
Throughput (IO/sec)	1473.03	1020.89	5534.83	6542.57	2263.9	72.9%	540.9%
Latency (ms)	6.71	7.89	14.40	4.46	3.0	44.2%	222.9%
Read latency (ms)	6.71	7.89	14.40	4.46	3.0	44.2%	222.9%
Postmark							
	ArrayA	**ArrayB**	**ArrayC**	**ArrayD**	**MAD**	**COV**	**Max. Diff.**
Bandwidth (MB/sec)	27.39	22.20	14.48	38.62	6.1	30.3%	166.7%
Throughput (IO/sec)	278.92	232.44	165.42	421.64	75.7	34.2%	154.9%
Latency (ms)	50.74	28.62	110.14	19.82	28.9	67.3%	455.7%
Write latency (ms)	75.90	38.05	175.20	23.71	48.5	75.6%	638.9%
Read latency (ms)	8.76	27.90	7.42	10.88	7.1	60.2%	276.0%
Cello							
	ArrayA	**ArrayB**	**ArrayC**	**ArrayD**	**MAD**	**COV**	**Max. Diff.**
Bandwidth (MB/sec)	8.60	4.33	5.04	10.03	2.1	30.4%	98.8%
Throughput (IO/sec)	807.82	435.87	481.05	974.33	216.3	33.3%	123.5%
Latency (ms)	19.10	44.86	42.51	7.40	15.2	55.5%	506.2%
Write latency (ms)	41.15	123.67	108.74	9.10	45.5	66.8%	1259.0%
Read latency (ms)	6.98	9.35	9.14	10.14	1.0	13.2%	45.3%
TPC-C							
	ArrayA	**ArrayB**	**ArrayC**	**ArrayD**	**MAD**	**COV**	**Max. Diff.**
Bandwidth (MB/sec)	1.69	1.12	1.96	2.40	0.4	25.9%	114.3%
Throughput (IO/sec)	211.60	140.04	243.84	296.92	47.3	25.5%	112.0%
Latency (ms)	5.78	17.29	5.12	1.94	4.9	77.2%	791.2%
Write latency (ms)	20.82	85.17	26.93	2.81	25.6	91.0%	2931.0%
Read latency (ms)	4.15	7.31	2.57	2.15	1.7	50.1%	240.0%

Table 7.3: Performance is measured for each of the application samples. The average value for each measurement is reported in this table. The mean absolute deviation (MAD), coefficient of variation (COV), and maximum relative difference quantify how the averages change across disk arrays.

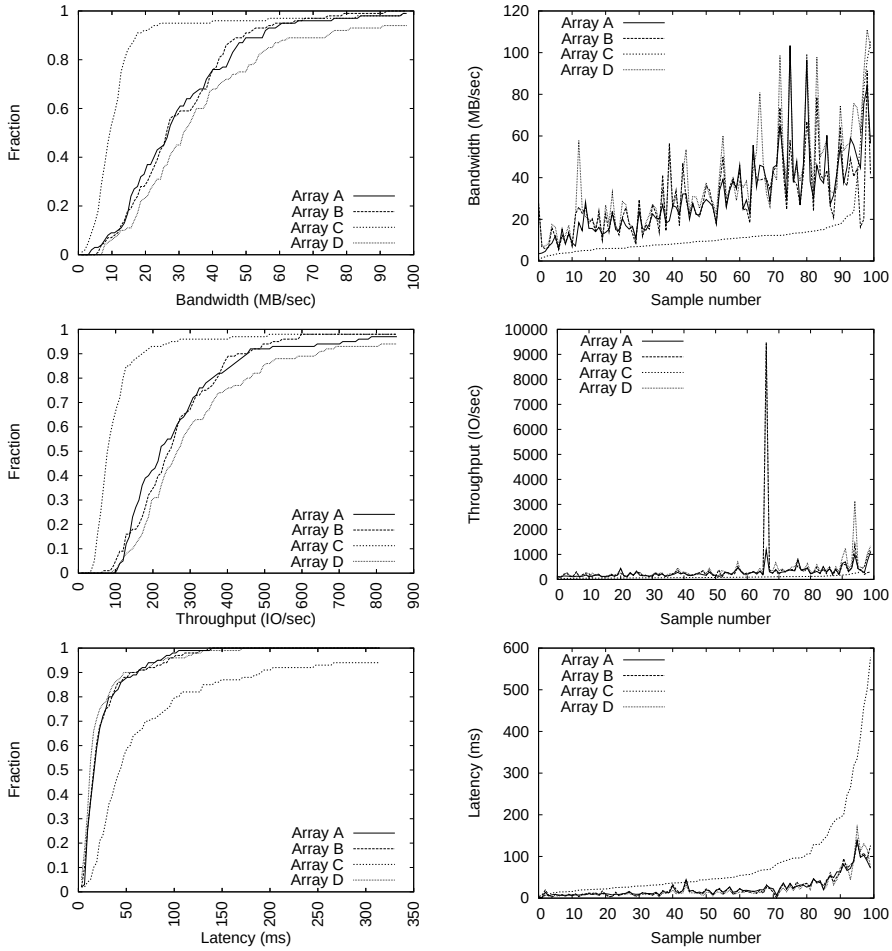

Figure 7.3: The cumulative distribution of performance for FitnessDirect is shown (left). In addition, the performance of each array is shown (right), sorted by the performance of Array C.

Looking now at the performance graphs in the right half of Figure 7.3, one can directly compare the performance behavior of the disk arrays. Again, each graph sorts the performance of each array according to the sort order of Array C. The samples of Array C are plotted in ascending order of performance, and the other three arrays are sorted in that same order. As with the CDFs, one can see that Arrays A, B, and D have very similar performance behavior. For example, a closer inspection of the bandwidth performance graphs indicates that Arrays A, B, and D track each other closely. They share similar discontinuities and spike at similar locations, only with different magnitudes.

Finally, Figure 7.4 plots the bandwidth relative fitness values of each disk array. Recall that a relative fitness value is simply the performance ratio of two arrays for a given workload sample: a value of 1.0 indicates that two arrays perform equally, and values other than 1.0 indicate a performance change when moving from one array to another. Therefore, the graphs illustrate how each pairing of arrays differs over the `FitnessDirect` samples. As can be seen in the graph, the fastest array on average (Array D) is not always the fastest array, as there are some relative fitness less than 1.0 when moving workload samples to that array from other arrays. For example, the relative fitness values of bandwidth for Array D when compared to Array A ("Array A to Array D" in Figure 7.4) indicate that some of the workload samples run slower on Array D than on Array A. This can be seen in the first bar of the probability distribution which shows a relative fitness value of less than 0.88 for 12% of the workload samples. In general, the objective of a relative fitness model is to predict these relative fitness values for a given workload sample and array pairing. The remaining PDFs in Figure 7.4 give one a sense of the range of relative fitness values that occur between different array pairings. Similar graphs exist for throughput and latency in the appendices.

In summary of `FitnessDirect`, one can make the following observations. First, Array D is the fastest array, on average, followed by Arrays A, B, and C. Second, Arrays A, B, and D exhibit similar distributions; Array C is an outlier. Third, there are a wide range of relative fitness values for each performance metric (i.e., the relative fitness of arrays depends on the characteristics of the workload sample).

`FitnessBuffered`

Continuing with `FitnessBuffered`, the application workloads are identical to `FitnessDirect`, but all I/O is directed through the OS buffer cache rather than directly to the disk arrays. As shown in Table 7.3, Array D is the fastest array (29 MB/sec), followed by Array A (28 MB/sec), Array B (21 MB/sec), and then Array C (16 MB/sec). The maximum relative difference in performance occurs between Arrays C and D, where average bandwidth varies by 78%, throughput by 86%, and latency by 165%.

Figure 7.5 shows the performance CDFs for each array. As was the case with `FitnessDirect`, Arrays A, B and D are somewhat clustered, and Array C is an outlier. This is especially evident in the latency CDFs. In the bandwidth and latency graphs (right half), one can see similarities among Arrays A, B, and D.

When compared to `FitnessDirect`, the performance differences among the arrays are not as extreme for `FitnessBuffered`. For example, Array D is twice as fast as Array C, on average. With `FitnessDirect`, Array D is over three times as fast. Again, the relative fitness of the arrays can change with the workload. In this case, the only differences among the samples are those introduced by the

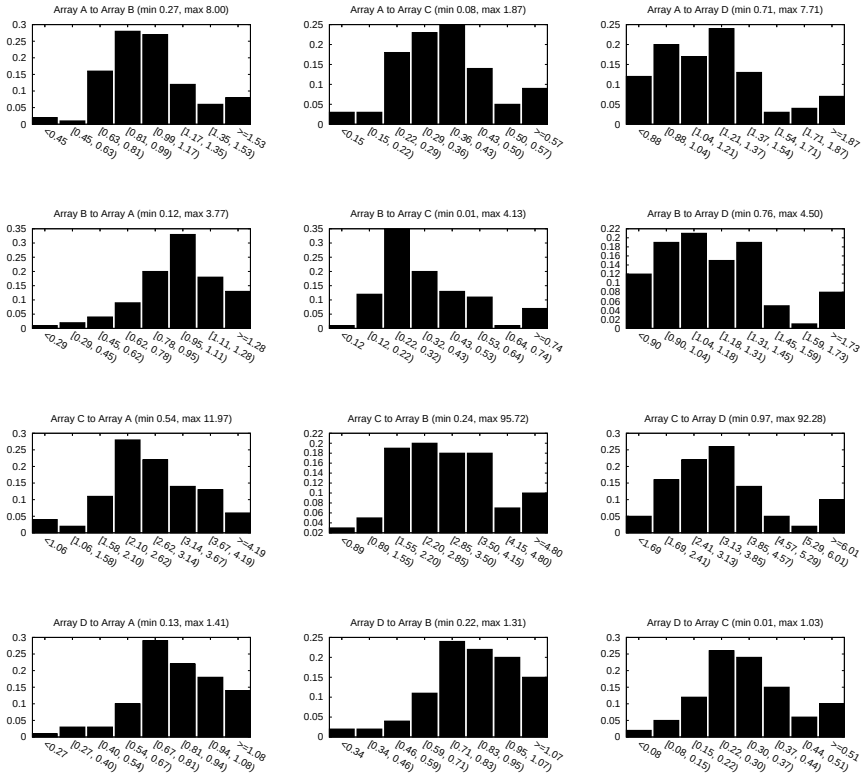

Figure 7.4: Each graph shows the probability distribution function (PDF) of the bandwidth relative fitness values for a given array pairing. The relative fitness value is the ratio of the performance of two arrays. Values greater than 1.0 indicate a performance increase as a workload is moved from array A to array B, and values less than one indicate a decrease. Note that the symmetry between $A \rightarrow B$ and $B \rightarrow A$ is not preserved in these graphs, due to the manner in which the PDFs are generated (i.e., different bucketing).

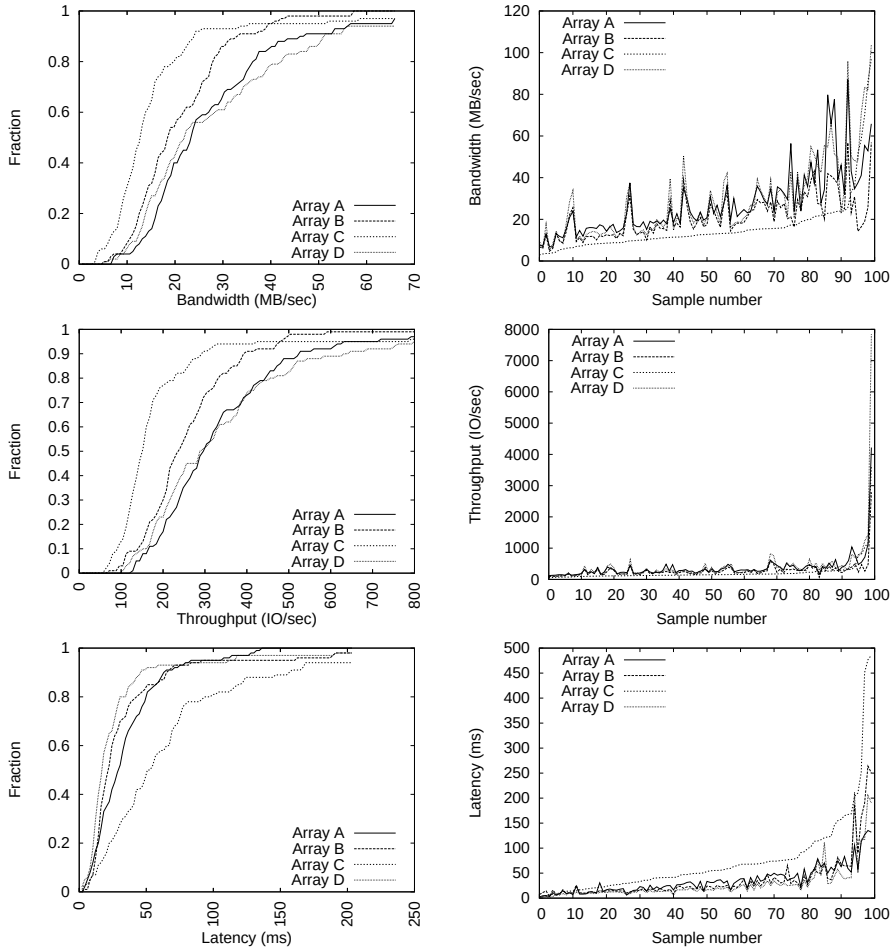

Figure 7.5: The cumulative distribution of performance for FitnessBuffered is shown (left). In addition, the performance of each array is shown (right), sorted by the performance of Array C.

buffer cache.

Just as with `FitnessDirect`, no one array is always slowest, or fastest, for `FitnessBuffered`. As can be seen in the relative fitness PDFs (see Appendices), there exists a wide range of relative fitness values for each pairing of arrays, which suggests that a single performance ratio between any two arrays would be a poor predictor of performance. For example, the bandwidth relative fitness of Array A to Array B ranges from 0.34 to 1.17, as indicated in the "min" and "max" values in the relative fitness PDFs. In other words, Array C can be up to 66% slower, or 17% faster than Array B, depending on the workload.

`FitnessFS`

The `FitnessFS` samples are identical to `FitnessDirect` and `FitnessBuffered`, but all I/O is directed through a file in a Linux file system (Ext2). From Table 7.3, Array D is again the fastest array (24 MB/sec), followed by Array A (23 MB/sec), Array B (18 MB/sec), and then Array C (15 MB/sec). The maximum relative difference in performance occurs between Arrays C and D, where average bandwidth varies by 55%, throughput by 56%, and latency by 179%. The performance graphs in Figure 7.6 illustrate still the similarity among Arrays A, B, and D, and the differences with Array C.

`FitnessCache`

For `FitnessCache`, Array D (40 MB/sec) is the fastest array, followed by Array C (35 MB/sec), Array A (16 MB/sec), and then Array B (16 MB/sec). Unlike the previous workloads, Array C is now the second fastest array, as opposed to the slowest, again underscoring the fact that the relative fitness of disk arrays is dependent on the workload. The maximum relative difference in performance for bandwidth and throughput now occurs between Arrays B and D, where average bandwidth varies by 156% and throughput by 541%. The maximum relative difference in latency is still between Arrays C and D (223%). Also, `FitnessCache` exhibits a different clustering of the disk arrays in terms of performance. The latency graph in Figure 7.7 (right half) is an especially nice example of how Arrays A, B, and D behave similarly, particularly for samples 25 through 35. Yet, in the bandwidth and throughput graphs, Arrays A and B are similar to one another, as are Arrays C and D.

`Postmark`

For `Postmark` (Figure 7.8), Array D is the fastest array (39 MB/sec), followed by Array A (27 MB/sec), Array B (22 MB/sec), and Array C (14 MB/sec). The maximum difference in performance occurs between Arrays C and D, where average bandwidth varies by 167%, throughput by 155%, and latency by 456%. As with previous workloads, the performance graphs shadow each other well in Figure 7.8. This is especially true for samples 5-35 in the latency graph (right half).

`Cello`

`Cello` (Figure 7.9) has much lower bandwidth when compared to the previous workloads. Array D is the fastest (10 MB/sec), followed by Array A (9 MB/sec), Array C (5 MB/sec), and Array B (4 MB/sec). The maximum difference in performance occurs between Arrays B and D, where average bandwidth varies by 99%, throughput by 124%, and latency by 506%. The cumulative distribution functions and

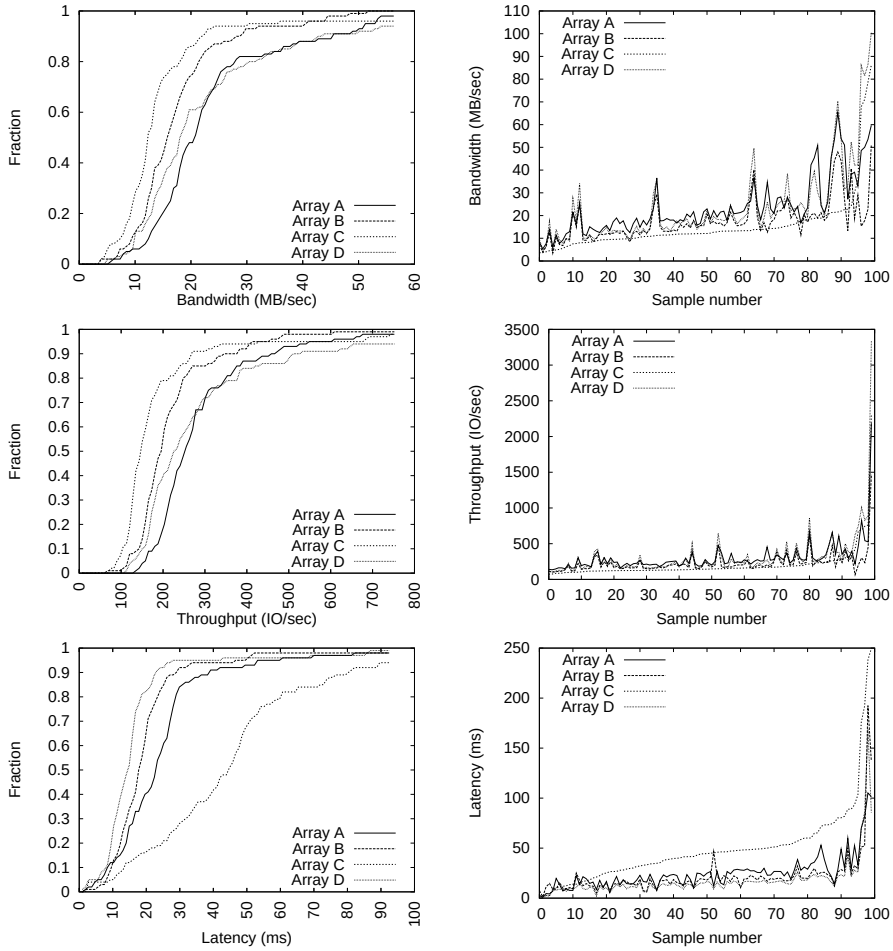

Figure 7.6: The cumulative distribution of performance for FitnessFS is shown (left). In addition, the performance of each array is shown (right), sorted by the performance of Array C.

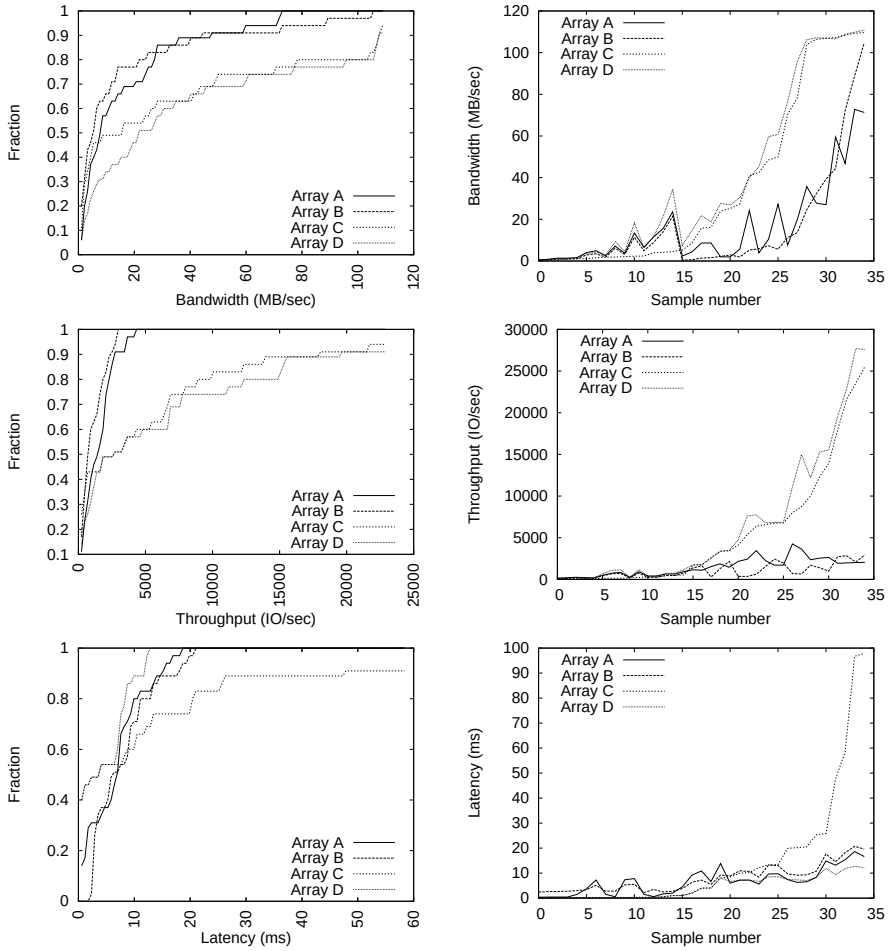

Figure 7.7: The cumulative distribution of performance for FitnessCache is shown (left). In addition, the performance of each array is shown (right), sorted by the performance of Array C.

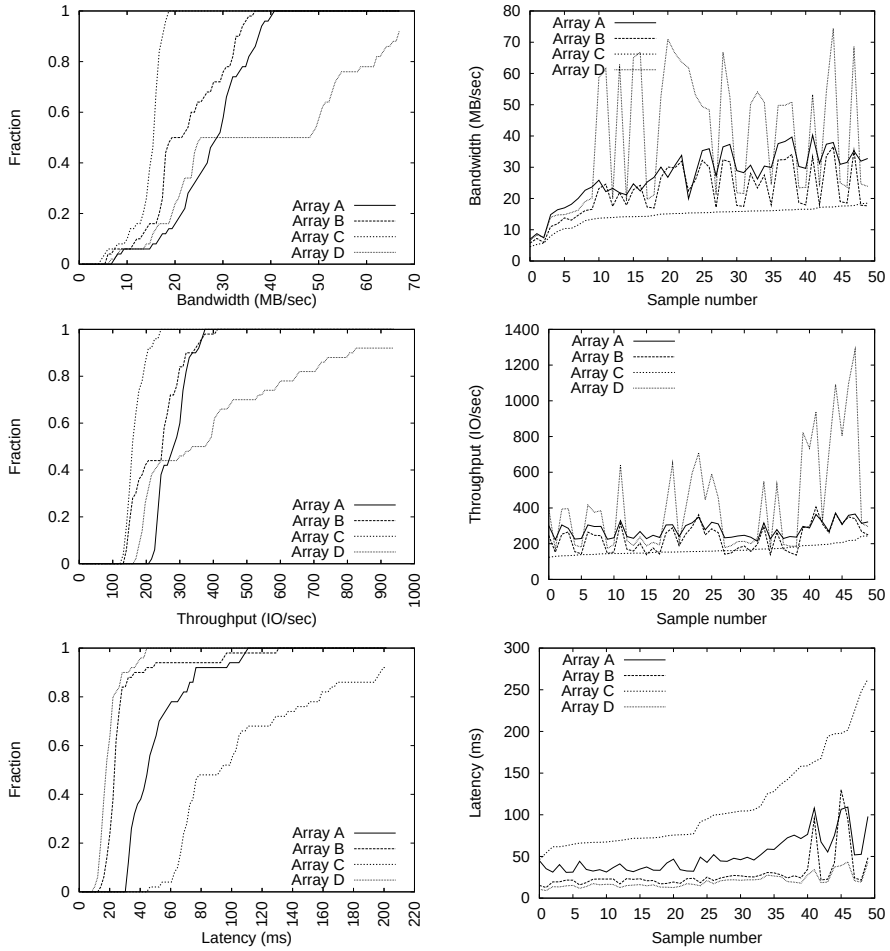

Figure 7.8: The cumulative distribution of performance for Postmark is shown (left). In addition, the performance of each array is shown (right), sorted by the performance of Array C.

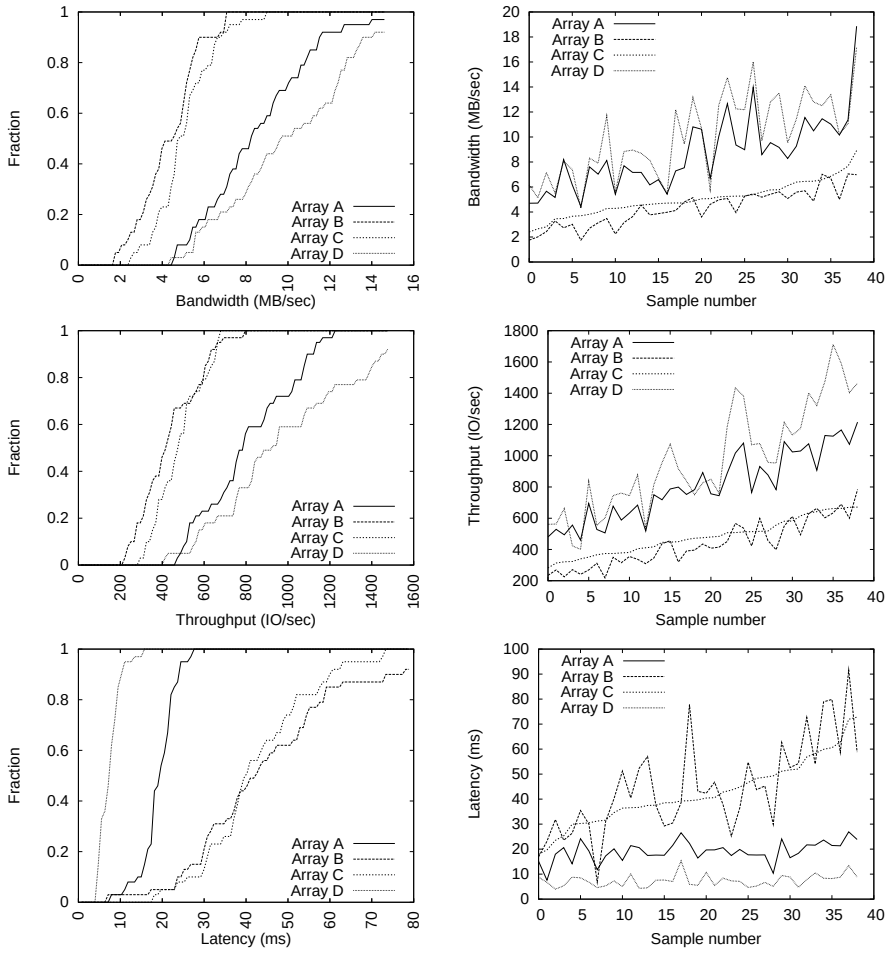

Figure 7.9: The cumulative distribution of performance for Cello is shown (left). In addition, the performance of each array is shown (right), sorted by the performance of Array C.

performance graphs in Figure 7.9 indicate a new clustering of arrays: Arrays A and D are similar, as are Arrays B and C. One can quickly glance at the CDFs and performance graphs in Figure 7.9 in order to see the two clusters.

The average intra-array performance for `Cello` varies by less than 10% for all but the bandwidth and throughput metrics on Array D. As shown by the Rel. Diff statistic in the Appendix for `Cello` training, the average bandwidth and throughput on Array D vary by 18% (i.e., the average bandwidth and throughput of the best and second-best iterations differ by 18%). Such variance must be considered when making predictions, as it limits the prediction accuracy one can achieve with a model.

`TPC-C`

`TPC-C` (Figure 7.10) has the lowest bandwidth of all workloads. Array D is the fastest (2.4 MB/sec), followed by Array C (2.0 MB/sec), Array A (1.7 MB/sec), and Array B (1.1 MB/sec). The maximum difference in performance occurs between Arrays B and D, where average bandwidth varies by 114%, throughput by 112%, and latency by 791%. Arrays A and B have very similar performance graphs, espe-

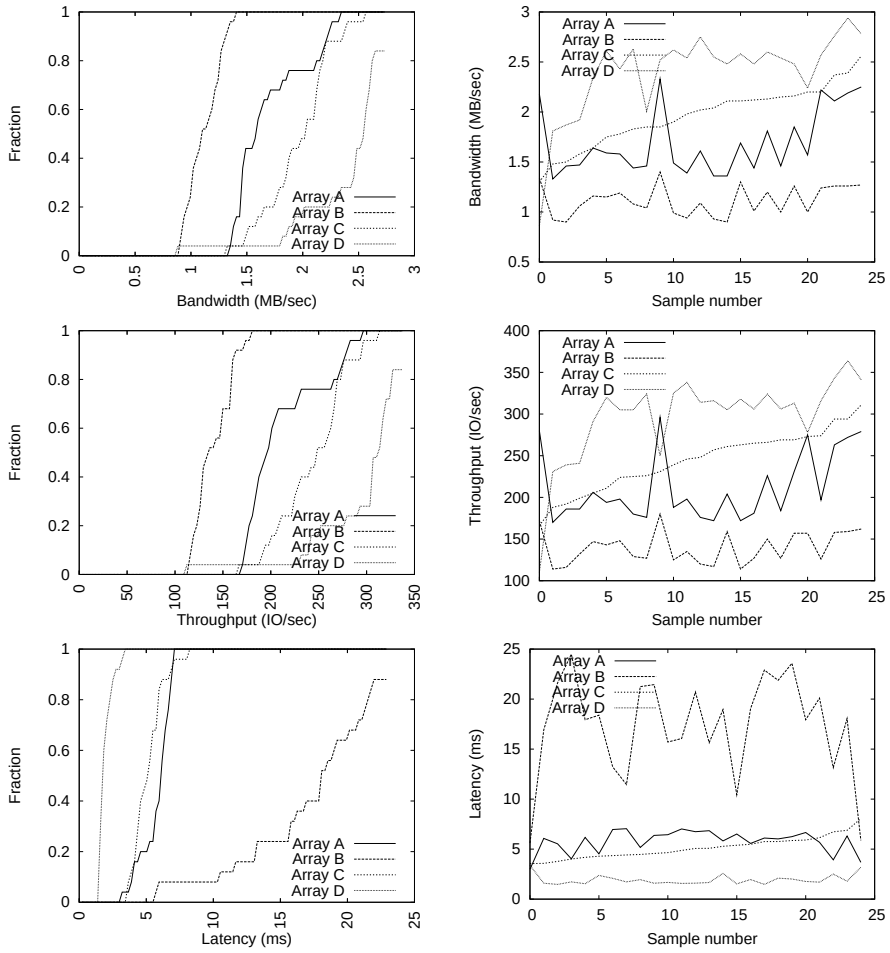

Figure 7.10: The cumulative distribution of performance for TPC-C is shown (left). In addition, the performance of each array is shown (right), sorted by the performance of Array C.

cially in the bandwidth and throughput graphs in Figure 7.10. For example, one can imagine multiplying the bandwidth or throughput curve of Array B by a constant performance ratio (a relative fitness value) to arrive at the performance curve of Array A.

Similarly to `Cello`, the variance in performance is low for most of the disk arrays. However, the average latency on Array B varies by 14%, as shown in the Appendix for the `TPC-C` training data.

Microbenchmarks

Across all application workloads evaluated, although one sees significant differences in the average performance across the disk arrays, one can also see similarities in the distributions of performance and in the shape of the sorted performance graphs. To illustrate this point even further, Figure 7.11 contains performance graphs for the set of simple microbenchmarks: sequential writes, random writes, sequential reads, and random reads. These workloads are a subset of the `FitnessDirect` samples. For each graph, only two workload characteristics are varied: the queue depth (from 1 to 64, stepping powers of 2) and the request size (from 1 to 32 KB, stepping powers of 2). Each graph is composed of 42 workload samples

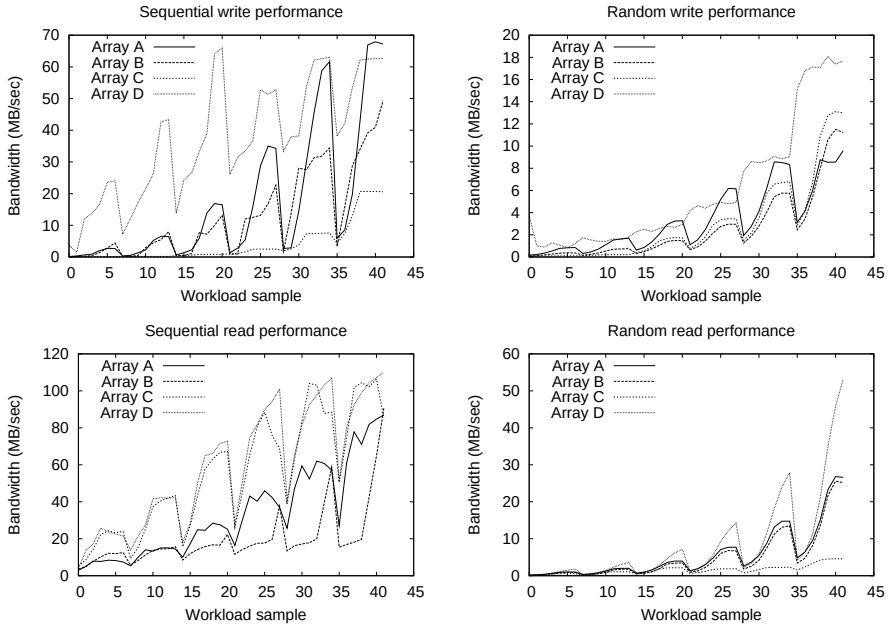

Figure 7.11: Each graph plots the performance of each disk array, for various queue depths and request sizes. There is one graph each for sequential writes, random writes, sequential reads, and random reads. In each graph, the queue depth varies from 1 to 64 (powers of two) and the request size from 1 to 32 KB (powers of two). The first 7 samples in each graph are for a request size of 1 KB and queue depths of 1 to 64, the second 7 samples are for a request size of 2 KB and queue depths of 1 to 64, and so on.

(7 queue depths $\times$ 6 request sizes). Unlike the sorted performance graphs, the samples are not sorted by the performance of Array C. Instead, because only two workload characteristics being varied, the samples are sorted first by request size and then by queue depth.

In looking at Figure 7.11, one can see how similar the performance graphs of the arrays are, despite the large differences in performance. This observation is the key behind relative fitness modeling. If arrays react similarly to workloads, the performance of one array may be a good predictor for another.

7.5.2 Workload characterization

Performance differences can lead to changes in workload characteristics.

This section summarizes the block-level workload characteristics for each application (i.e., from each array's point of view) and the extent to which they change across disk arrays. Changes in the workload characteristics are quantified in the same manner as the performance metrics. For each application, the average value for each workload characteristic (across all samples) is compared just as the performance averages were in the previous section, using the mean absolute deviation (MAD), the coefficient of variation (COV), and the maximum relative difference between averages (Max. Diff.). For example, the average write request size of `Postmark` is calculated for each of the disk arrays, across all of the `Postmark` samples. Then, the COV, MAD, and Max. Diff. statistics are used to quantify the change in average write request size across arrays. See Table 7.4.

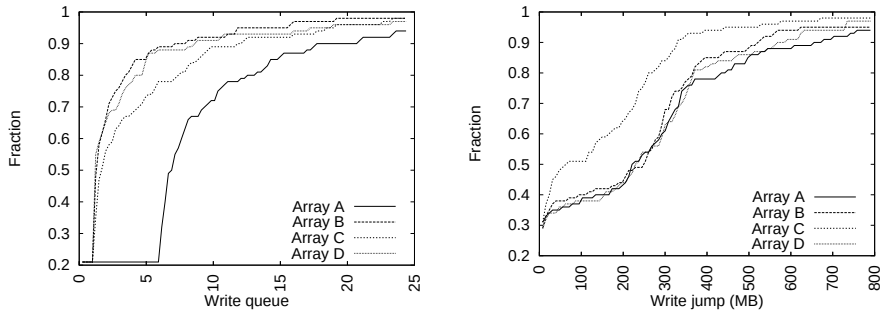

Figure 7.12: CDFs of the average write queue depth for the `FitnessDirect` samples (left half of figure) and the average write randomness for the `FitnessBuffered` samples (right half).

`FitnessDirect`

Beginning with `FitnessDirect`, the only workload characteristics that experience any change are the write and read queue depths. Of course, this is due to the different service rates of the disk arrays. Recall, `FitnessDirect` issues I/O directly to the arrays, so I/O requests are not subject to modification in a file system or buffer cache. Therefore, the queue depths are the only workload characteristics eligible for change. From Table 7.4, Array A has the longest write queues for the `FitnessDirect` samples (9 IOs on average), followed by Array C (4), Array D (3.4) and Array B (3.0). The maximum relative difference between these averages is 188%, which occurs between Arrays A and B. Similarly, the read queue depth varies by up to 57%.

To visualize the write queue depth differences, Figure 7.12 (left half) plots the cumulative distribution functions of the observed write queue depth, over each the 100 `FitnessDirect` training samples. One can see the similarity among Arrays B, C, and D. Array A is an outlier (it has the longest queues).

As was done with the performance measurements, to ensure that the workload changes are due to changes in the performance of the disk arrays, rather than variance, one can compare the Rel. Diff. value in the Appendix for `FitnessDirect` with the Max. Diff. value in Table 7.4. For example, the average write queue depth for `FitnessDirect` varies by less than 1% across different iterations of the same set of samples, when the samples are measured on the same disk array — compared to 188% when comparing the average queue depth across arrays. Throughout the rest of this section, unless otherwise stated, one can assume that the intra-array variance of a workload characteristic is much less than the inter-array variance, thereby indicating that the change in workload characteristics is due to the performance differences among the disk arrays.

`FitnessBuffered`

Looking now at `FitnessBuffered`, one can see the effects of issuing I/O through the OS buffer cache. Most notably, the maximum relative difference in the average write randomness is 28%, which occurs between Arrays B and C. In other words, Array C experiences write jumps that are, on average, 28% longer than those seen by Array B. Figure 7.12 (right half) illustrates the change, where the distribution of the write jump distance is markedly different for Array C. The other workload characteristics vary by less than 10%, as shown in Table 7.4.

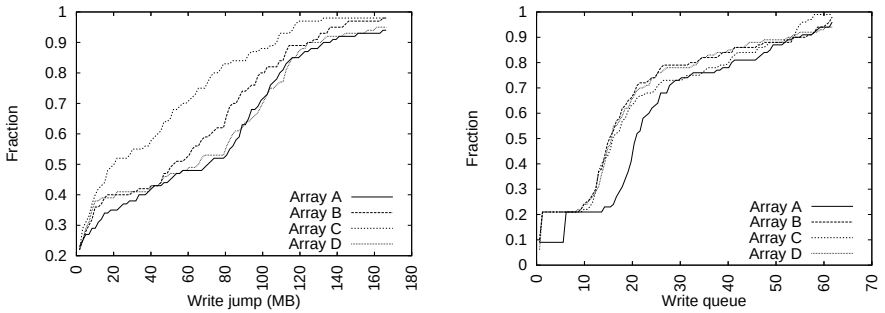

Figure 7.13: CDFs of the average write randomness and queue depth for the `FitnessFS` training samples.

Note that for the `FitnessBuffered` samples, Array C itself experiences considerable intra-array variance in the write randomness. As shown in the Appendix for `FitnessBuffered` training, the write randomness of Array C varies by 136%, across different iterations of the same set of samples. This is a case where the intra-array variance of a workload characteristic (Rel. Diff.) is not significantly less than the inter-array variance (Max. Diff.). Therefore, it is difficult to conclude, using only the Rel. Diff. statistic, that a change in workload characteristics is due to performance differences among the arrays — because the workload characteristics change even on the same array. In general, the average write randomness is the characteristic that experiences the most intra-array variance, which suggests that it might be too sensitive a workload characteristic. One possible remedy for future work is to use the median jump distance, or a percentile, rather than the mean. Nonetheless, one can see in Figure 7.12 (right half) that the distribution of write randomness for Array C is noticeably different from the other arrays.

`FitnessFS`

`FitnessFS` issues I/O through the file system and behaves similarly to `FitnessBuffered`, in that Array C is an outlier with respect to the spatial randomness of writes. The largest difference occurs between Array C and Array D, where Array D experiences write jumps that are, on average, 179% longer than those seen by Array C. In addition to the write randomness, the average write queue depth varies by up to 25%. Figure 7.13 illustrates.

`FitnessCache`

The workload characteristics of `FitnessCache` (a read-only workload) vary only slightly. The only measurable change is for the average read queue depth, which varies by up to 21%.

`Postmark`

One sees a variety of change with `Postmark`. The average write queue depth varies by up to 63%, the read queue depth by 45%, and the write randomness by 17%. Again, Array C sees the least randomness. Figure 7.14 illustrates.

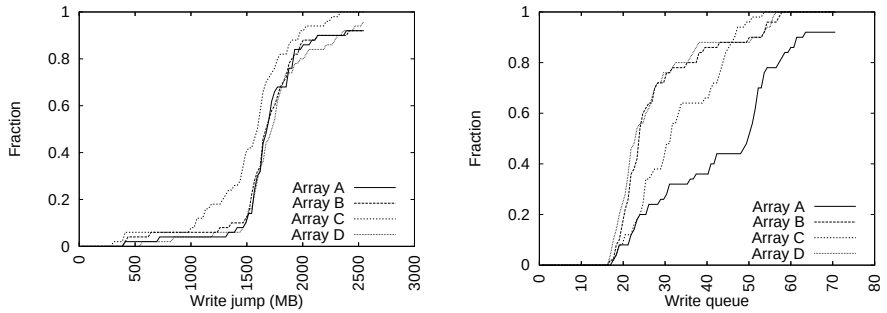

Figure 7.14: CDFs of the average write randomness and queue depth for the `Postmark` training samples.

`Cello`

The write randomness of `Cello` varies up to 44% and the write queue depth up to 16%. Unlike the other workloads, one also sees a significant change in the write request size. For example, the average write request size varies from 18 KB on Array C to 21 KB on Array D.

`TPC-C`

The write queue depth of `TPC-C` varies up to 132% and the write randomness up to 105%. Even the average write fraction experiences a 50% swing (e.g., from 16% writes on Array C to 24% writes on Array D).

7.5.3 Summary

Workload characteristics can change across disk arrays when the arrays have different performance characteristics. The write and read queue depths can obviously change, as disk arrays may have different service rates. Other measurable changes can be seen in the spatial randomness, request sizes, even the read/write ratio. In support of Hypothesis 1, Experiment 1 is summarized as follows:

Result 1 *When the same workload is run on different disk arrays, performance varies considerably. On average, bandwidth varies up to 189% (`FitnessDirect`), throughput up to 541% (`FitnessCache`), and latency up to 791% (`TPC-C`). Moreover, there are measurable changes in the workload characteristics. On average, the write queue depth varies up to 188% (`FitnessDirect`), the write randomness up to 179% (`FitnessFS`), the read queue depth up to 57% (`FitnessDirect`), the write fraction up to 50% (`TPC-C`), and the write size up to 21% (`Cello`). Insignificant changes are seen in the read randomness and the read size.*

In general, the write characteristics vary more than the read characteristics, as read requests are not delayed, aggregated, and coalesced within the page cache in the same manner that write requests are. The write characteristics that change the most are the spatial randomness and queue depth. In particular, a slower disk array can experience less spatial write randomness than a faster array. Such is the case with Array C.

	ArrayA	ArrayB	ArrayC	ArrayD	MAD	COV	Max. Diff.
FitnessDirect							
Write fraction	0.48	0.48	0.48	0.48	0.0	0.2%	0.0%
Write size (KB)	102.18	102.12	102.40	102.06	0.1	0.1%	0.3%
Read size (KB)	119.88	119.54	120.06	119.65	0.2	0.2%	0.4%
Write jump (MB)	2421.29	2467.32	2445.39	2432.69	14.7	0.7%	1.9%
Read jump (MB)	2983.15	2973.14	3025.42	2979.56	17.6	0.7%	1.8%
Write queue	8.69	3.02	4.37	3.47	1.9	46.0%	187.8%
Read queue	3.13	4.92	3.51	4.37	0.7	17.6%	57.2%
FitnessBuffered							
Write fraction	0.31	0.31	0.30	0.31	0.0	1.8%	3.3%
Write size (KB)	99.62	100.34	104.30	99.77	1.6	1.9%	4.7%
Read size (KB)	63.82	63.94	63.77	63.77	0.1	0.1%	0.3%
Write jump (MB)	255.22	237.39	302.71	252.20	20.4	9.4%	27.5%
Read jump (MB)	2057.00	2061.67	2116.93	2055.77	22.0	1.2%	3.0%
Write queue	26.36	26.24	24.97	25.96	0.5	2.1%	5.6%
Read queue	5.07	5.07	4.79	5.06	0.1	2.4%	5.8%
FitnessFS							
Write fraction	0.30	0.30	0.30	0.30	0.0	0.6%	0.0%
Write size (KB)	104.56	106.12	109.11	105.15	1.4	1.6%	4.3%
Read size (KB)	59.72	59.59	59.30	59.10	0.2	0.4%	1.1%
Write jump (MB)	71.10	55.13	39.51	110.04	21.6	38.0%	178.5%
Read jump (MB)	630.12	629.82	636.29	638.64	3.8	0.6%	1.4%
Write queue	25.12	20.08	21.22	20.34	1.7	9.3%	25.1%
Read queue	2.87	2.83	2.91	2.83	0.0	1.1%	2.8%
FitnessCache							
Read size (KB)	15.77	15.77	15.77	15.77	0.0	0.0%	0.0%
Read jump (MB)	1855.03	1861.46	1856.29	1850.03	3.2	0.2%	0.6%
Read queue	6.76	6.68	5.70	5.59	0.5	8.7%	20.9%
Postmark							
Write fraction	0.76	0.77	0.76	0.77	0.0	0.7%	1.3%
Write size (KB)	110.72	109.48	108.27	109.50	0.6	0.8%	2.3%
Read size (KB)	53.72	54.12	51.53	54.11	0.9	2.0%	5.0%
Write jump (MB)	1754.38	1722.18	1518.64	1771.54	86.5	6.0%	16.6%
Read jump (MB)	3654.38	3631.52	3784.66	3558.88	63.6	2.2%	6.3%
Write queue	44.46	28.20	32.88	27.34	5.6	20.5%	62.6%
Read queue	2.23	1.95	1.66	1.54	0.2	14.5%	44.8%
Cello							
Write fraction	0.45	0.44	0.44	0.47	0.0	2.8%	6.8%
Write size (KB)	17.70	21.41	17.67	20.69	1.7	8.8%	21.2%
Read size (KB)	6.98	7.14	7.37	7.22	0.1	1.9%	5.6%
Write jump (MB)	8.64	8.79	9.15	12.44	1.3	16.0%	44.0%
Read jump (MB)	313.28	303.72	301.67	326.10	8.5	3.1%	8.1%
Write queue	57.95	51.73	53.74	59.93	3.1	5.8%	15.8%
Read queue	5.99	5.84	5.76	6.08	0.1	2.1%	5.6%
TPC-C							
Write fraction	0.18	0.21	0.16	0.24	0.0	14.1%	50.0%
Write size (KB)	8.52	8.45	8.61	8.77	0.1	1.4%	3.8%
Read size (KB)	8.14	8.12	8.17	8.12	0.0	0.2%	0.6%
Write jump (MB)	125.28	208.12	101.56	132.96	33.1	28.1%	104.9%
Read jump (MB)	273.16	270.48	277.00	299.40	9.7	4.1%	10.7%
Write queue	18.96	30.82	13.28	13.52	5.8	37.2%	132.1%
Read queue	1.00	1.00	1.00	1.00	0.0	0.1%	0.0%

Table 7.4: Workload characteristics are measured for each application. Average values are reported with their mean absolute deviation (MAD), coefficient of variation (COV), and maximum relative difference.

7.6 Experiment 2: absolute models

Experiment 1 showed that block-level workload characteristics can change across disk arrays. Therefore, an absolute model of array i may experience an increase in prediction error when the workload characteristics are obtained from a different array j (Hypothesis 2). Of course, as described in Chapters 2 and 5, this is how performance models are to be used in practice.

To test Hypothesis 2, absolute models are built for each disk array, as described in Section 6.3. Summarizing from Section 7.4.3 (Table 7.2), the performance metrics (dependent variables) include the bandwidth, throughput, and latency of each sample; and the workload characteristics (predictor variables) include the write fraction, write request size, read request size, write randomness, read randomness, write queue depth, and read queue depth.

Regarding queue depth, it is arguably more a resource utilization or performance metric than a workload characteristic and, as such, should not be used in the construction of an absolute model (which discards performance and utilization information). However, the absolute models evaluated in this work, overall, are more accurate when the queue depth is included. The queue depth is an approximation of the true multi-programming level of the application — a valuable, but difficult, workload characteristic to obtain. So, it can be better to use an informative workload characteristic that changes, rather than discard it altogether. Indeed, if one were to discard all the workload characteristics that have the potential for change, as per the result of Experiment 1, there would be few workload characteristics left for training the models!

To begin, performance models are built and evaluated for each application. Such *per-application* models are used to illustrate the differences among the applications, specifically which workload characteristics are the best predictors of performance. In addition, they present somewhat of a best-case scenario for all the models, where the training data is representative of the testing data.

For the per-application models, 50% of the collected samples are used to train the models and the remainder are used for testing. In addition, various training/testing splits are explored to compare how quickly the models learn (i.e., models are first trained on 25% of the samples and tested on 75%, then trained on 30% of the samples and tested on 70%, and so on).

After the per-application models are presented, more desirable *mixed* models are trained by combining 50% of all application samples. These models are more desirable in the sense that they attempt to learn a disk array's performance across a wide range of applications. This is a test to see that a single black-box can learn to predict the performance of different applications. All of the per-application and mixed models can be found in the regression tree appendices. Each model is rendered using the *dot* programming environment for directed graphs [9]. Selected models are included throughout the evaluation for illustrative purposes.

To test the models, three predictions (bandwidth, throughput, and latency) are made for each sample across all pairings of disk arrays (recall that the models are directional). Because there are four arrays in this evaluation, there are 16 permutations of two (4 where $i = j$ and 12 where $i \neq j$). Therefore, for each sample, absolute models are used to make 16 bandwidth predictions, 16 throughput predictions, and 16 latency predictions. Stated differently, four predictions are made for each performance metric on each disk array, for each sample: one uses the workload characteristics as measured by array i, and the

	Bandwidth				Throughput				Latency			
Absolute	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
ArrayA	0.26	0.27	0.23	0.18	0.20	0.26	0.23	0.29	0.18	0.44	0.58	0.77
ArrayB	0.34	0.17	0.27	0.23	0.23	0.22	0.24	0.29	0.27	0.26	0.33	0.33
ArrayC	0.32	0.19	0.23	0.18	0.27	0.26	0.23	0.29	0.30	0.32	0.28	0.33
ArrayD	0.32	0.17	0.23	0.22	0.24	0.22	0.24	0.29	0.27	0.31	0.31	0.34

Table 7.5: Median relative error for `FitnessDirect`.

other three use the workload characteristics as measured by a different array j. As such, the errors of these four predictions can be compared to see if changes in the workload characteristics affect prediction accuracy.

Of course, as discussed, characterizing a workload on array i obviates the need for a performance prediction of array i. Nonetheless, the $i = j$ prediction serves as a baseline for the absolute model, where workload characteristics experience no change as a workload is moved from one array to another. As such, it is used to calculate the increased error due to changes in the workload characteristics.

Models are quantitatively compared by their median relative error. In addition, two visual comparisons are used: the cumulative distribution function (CDF) of their relative error, and a graph of the median relative error for the various training/testing splits. From the CDFs, one can also obtain any of the percentiles (e.g., the 90th percentile of relative error).

Note, the mean relative error is not reported in the evaluation but is contained in the appendices. For the applications and disk arrays evaluated, the mean error exaggerates modeling error (due to outliers) and is a misleading indicator of a model's accuracy. Also not contained in the evaluation, but contained in the appendices, are probability distribution functions (PDFs) of the relative errors for each model.

7.6.1 Per-application models

Prediction error increases as a result of changing workload characteristics.

The prediction errors of the absolute models are now presented. Again, special attention is given to `FitnessDirect` so as to provide instruction on interpreting the tables and figures.

`FitnessDirect`

There are 200 `FitnessDirect` samples (100 for training the absolute models and 100 for testing). The median relative error when predicting the performance of the 100 test samples for each pairing of arrays is shown in Table 7.5. The median relative error across all pairings is shown in Figure 7.15.

For example, the median relative error of the Array A latency model, when using the characteristics obtained from Array A, is 18% (i.e., 50 of the predictions had a relative error of 18% or less). Through a quick inspection of the table, one can see that the least error usually occurs along the diagonals, that is, for the pairings where $i = j$, in which a workload is characterized on the same array for which the prediction is being made. Unfortunately, this is not a realistic scenario, so one must be prepared for some amount of error due to changing workloads.

The values off the diagonal ($i \neq j$) show the impact changing workload characteristics can have on prediction accuracy. As described in Chapter 5, the absolute model for array i is trained with workload

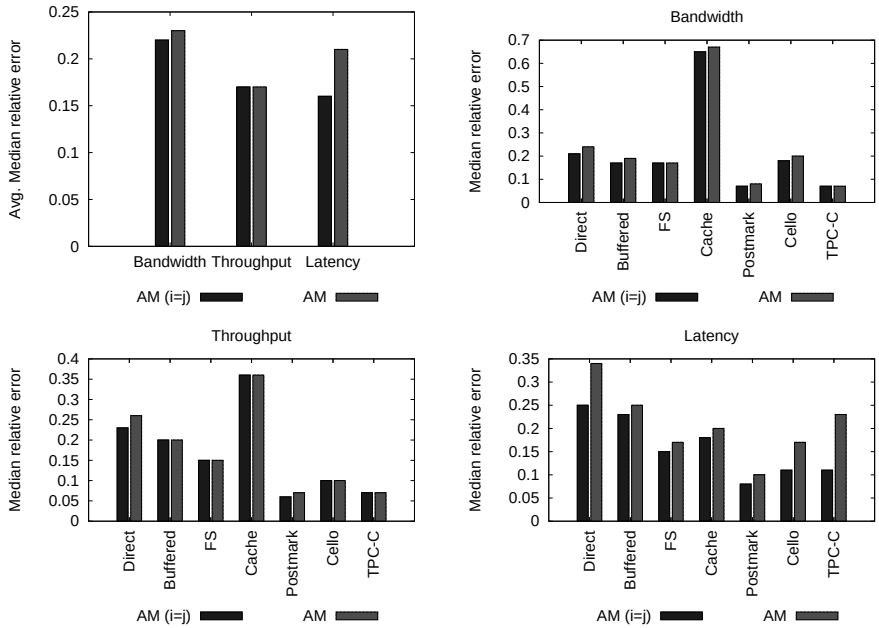

Figure 7.15: Median relative error: Experiment 2 (per-application models). The graphs show the median relative error of each application, comparing the idealized absolute model, title "AM (i=j)," with the absolute model (AM). Also shown are the average median relative errors of the applications for each performance metric.

characteristics as measured by array i, not a different array j, hence the increases in error. As examples, the median relative error for the Array A latency model increases from 18% to 27% when the workload characteristics are obtained from Array B, 30% when obtained from Array C, and 27% from Array D. Similarly, the error of the absolute throughput model for Array A is 20% when the workload characteristics are obtained from Array A, increasing to 23% when obtained from Array B, 27% from Array C, and 24% from Array D. For bandwidth, the prediction error increases from 26% when characteristics are obtained from Array A to 34% from Array B, 32% from Array C, and 32% from Array D. Overall, the median relative error of the bandwidth predictions is 20% for the idealized absolute models, and this increases to 23% for the in-practice absolute models. Similarly, the throughput prediction error increases from 23% to 26%, and the latency prediction error increases from 25% to 34%.

To visualize the differences in prediction error, Figure 7.16 plots the cumulative distribution of relative error for all pairwise predictions. Each graph compares the idealized absolute models with the in-practice absolute models. In these graphs, the darkest line (least error) is that of the idealized absolute model. For `FitnessDirect` (top row), one can see the increase in error due to changing workloads, especially in the latency graph.

Finally, the accuracy of the idealized and in-practice absolute models is compared across varying levels of training. The models begin by training on 25% of the samples (testing on the remaining 75%) and proceed to a 75/25 split in increments of 5%. Figure 7.17 plots the median relative error across all such splits.[1] As can be seen in the graph, the idealized absolute, titled "AM ($i = j$)," is more accurate than

[1]Note, the errors in the error-versus-training graph, particularly when training on 50% of the samples and testing on

Overall		Bandwidth		Throughput		Latency	
Predictor	**Score**	**Predictor**	**Score**	**Predictor**	**Score**	**Predictor**	**Score**
Read queue	1.00	Read queue	1.00	Read queue	1.00	Write queue	1.00
Write queue	0.76	Read size	0.63	Write size	0.80	Read queue	0.86
Read jump	0.44	Read jump	0.56	Write jump	0.71	Read size	0.24
Read size	0.41	Write size	0.54	Read jump	0.61	Read jump	0.23
Write size	0.38	Write queue	0.44	Write fraction	0.49	Write jump	0.10
Write jump	0.27	Write jump	0.27	Write queue	0.43	Write fraction	0.08
Write fraction	0.14	Write fraction	0.05	Read size	0.25	Write size	0.07

Table 7.6: The normalized importance measure of each predictor for `FitnessDirect`.

the in-practice absolute model, titled "AM," across a variety of training/testing splits. Of course, no amount of training can teach an absolute model how to deal with changing workloads.

In support of Hypothesis 2, one sees that changing workloads increase the prediction error of the `FitnessDirect` models. In hindsight, this should not come as a surprise. A simple inspection of the workload characteristics and an analysis of the regression trees indicate that the most likely to change workload characteristics are often those that also provide the most information.

Table 7.6 ranks the workload characteristics from most information to least. Again, the rankings are based on an analysis of the regression tree models. The attribute with the highest ranking is that involved in the most splits; it has a value of 1.0. For example, the best overall predictors of `FitnessDirect` latency are the write and read queue depths. Therefore, one can logically conclude that a change in these characteristics will lead to an incorrect traversal of the trees. Figure 7.18 shows the actual regression trees for the absolute latency models, illustrating how a change in the write queue depth can result in an inaccurate prediction.

The remainder of this section provides further support of Hypothesis 2 from the remaining applications. For each application, only the cumulative distribution functions of error and the error-versus-training graphs are shown. The pairwise median relative errors and the attribute importance measures are summarized from the appendices.

`FitnessBuffered`

Recall that the workload characteristic experiencing the most change for `FitnessBuffered` is the spatial randomness of writes (Table 7.4). In particular, the maximum relative difference in the average write randomness across arrays is 28%. Such change suggests that any model based on this characteristic is likely to experience error. However, the write randomness is not a strong predictor of performance. For `FitnessBuffered`, the strongest predictors of performance are the read characteristics (read queue depth, read randomness, write fraction, and read request size). Fortunately, for an absolute model, these workload characteristics experience only slight change across the arrays. As a result, the increase in error due to changing workload characteristics is minor. Figure 7.16 plots the cumulative distribution of error. Unlike the error distributions for `FitnessDirect`, little difference can be seen between the

the remainder, cannot be directly compared to the median relative errors in Figure 7.23. Although both graphs report error when the training/testing split is 50%, the error-versus-training graph plots the *average* median relative error of the array pairings, as opposed to Figure 7.23 which plots median relative error when the predictions from all array pairings are combined. The resulting values are similar, but not necessarily the same.

idealized and in-practice absolute models. The median relative error increases by only 1%. Still, small differences can be observed in Figure 7.17, where the average median relative error is shown for various training/testing splits. In general, as with `FitnessDirect`, the most accurate predictions occur when a workload is characterized on the same array for which a prediction is being made.

`FitnessFS` and `FitnessCache`

`FitnessFS` and `FitnessCache` are also affected only slightly by changing workloads. Again, this is due to the fact that the best predictors of performance for these workloads are the read I/O characteristics, which only change slightly across the disk arrays for these workloads. Specifically, the best performance predictors for `FitnessFS` are the write fraction, followed by the read queue depth and the read randomness. For `FitnessCache`, they are the read randomness, followed by the read queue depth and the read request size.

In the cumulative error distributions in Figure 7.16, very little difference can be seen between the idealized and in-practice absolute models. The difference in median relative error is less than 3% for both of these workloads. One can see these small differences in the error-versus-training graphs in Figure 7.17. In nearly all cases, due to the magnifying effect of the figure, the idealized absolute models are distinguishable from the absolute models and are more accurate.

`Postmark`

With `Postmark`, the best predictors are the write characteristics. The write fraction is the best overall predictor, followed by the write request size. However, neither of these characteristics experience much change across arrays. Therefore, the error introduced by changing workloads is also small for `Postmark`, though more evident than the error for `FitnessFS` and `FitnessCache`.

In the cumulative error distributions in Figure 7.16, one can see the difference between the idealized and in-practice absolute models for the throughput and latency predictions. However, the difference in median relative error between the idealized and in-practice absolute models is still less than 3% for all performance metrics . These differences can be seen in the error-versus-training graphs in Figure 7.17. In all cases, the idealized absolute models are distinguishable from the in-practice models and are more accurate.

`Cello`

The effects of changing workloads on the `Cello` predictions are much more pronounced, particularly for the latency predictions. The median relative error of the latency predictions increase from 11 to 17%; the changes in bandwidth and throughput are minor. The differences can be seen in the error distributions graphs in Figure 7.16 and the error-versus-training graphs in Figure 7.17.

As can be seen in the throughput graph in Figure 7.17, the idealized absolute model actually does worse than the in-practice model, after training on 45% and 60% of the samples. Such inconsistencies in the results appear now and again. In general, regression trees can be sensitive to the amount of training data and may therefore experience sudden fluctuations in accuracy. Often, the fluctuations are shared across all models after training on some number of samples. However, there are cases where only some of the models experience a sudden change in accuracy.

Overall		Bandwidth		Throughput		Latency	
Predictor	**Score**	**Predictor**	**Score**	**Predictor**	**Score**	**Predictor**	**Score**
Read queue	1.00	Read queue	1.00	Read queue	1.00	Write fraction	1.00
Write fraction	0.72	Read jump	0.70	Read jump	0.66	Read queue	0.72
Read jump	0.56	Read size	0.54	Write size	0.50	Write queue	0.39
Read size	0.40	Write size	0.51	Read size	0.46	Read jump	0.24
Write size	0.37	Write fraction	0.49	Write queue	0.24	Read size	0.16
Write queue	0.37	Write queue	0.30	Write jump	0.21	Write size	0.09
Write jump	0.19	Write jump	0.29	Write fraction	0.19	Write jump	0.05

Table 7.7: The normalized importance measure of each predictor for the absolute models.

For `Cello`, the best latency predictors are the write fraction and the read queue depth. However, as shown in Section 7.5, these workload characteristics experience change across the arrays. To cite a specific example, due to changes in the write request sizes, the write fraction of sample #67 decreases from 44% to 37% when moving from Array D to Array B. Consequently, an incorrect path is taken in the tree. Figure 7.19 shows the latency model for Array B. If one inputs the write fraction as measured by Array D (0.44) into this model, the predicted latency is 58.7 ms. However, when inputing the write fraction as measured by Array B (0.37), the predicted latency is 46.8 ms. The actual latency for sample #67 on Array B is 49.61 ms. Therefore, the idealized absolute model has a relative error of 6% versus an error of 18% for the in-practice model. This is one specific example of how a relatively small change in the workload characteristics can lead to an increase in prediction error. Overall, the most accurate `Cello` predictions occur when the workload characteristics are obtained from the same array for which the prediction is being made.

`TPC-C`

Of all the workloads, the `TPC-C` latency predictions are affected the most by changing workloads. The best predictors of `TPC-C` latency are the write queue depth and the write fraction, and both show considerable change across disk arrays. Recalling from Table 7.4, the average write queue depth changes by 132% between Array B (31 outstanding) and Array C (13 outstanding), and the write fraction changes by 50% between Array C (16% writes) and Array D (24%). Consequently, the median relative error increases from 11% to 23%. The differences can be seen in the error distribution graphs in Figure 7.16 and the error-versus-training graphs in Figure 7.17. Although the bandwidth and throughput prediction errors only increase slightly, the error-versus-training graphs distinguish the in-practice and idealized absolute models, and the idealized model is more accurate across all training/testing splits for each performance metric.

7.6.2 Best predictors

Table 7.7 summarizes the best overall predictors for bandwidth, throughput, and latency. The values shown represent averages over all of the absolute models. The best predictors for bandwidth and throughput are the read characteristics, specifically the read queue depth and read randomness. For latency, however, the best predictors are the write fraction and the read and write queue depths.

Application	Per-application	Mixed-model
FitnessDirect	0.28	0.25
FitnessBuffered	0.21	0.19
FitnessFS	0.16	0.17
FitnessCache	0.40	0.31
Postmark	0.08	0.09
Cello	0.15	0.18
TPC-C	0.10	0.12

Table 7.8: Per-application versus mixed model median relative error for the in-practice absolute models.

7.6.3 Mixed models

Mixed models are built using 50% of all application samples. Three models are built for each disk array (bandwidth, throughput, and latency). In general, the trees are much larger for the mixed models. This is because the applications, combined, cover a wider range of workload characteristics and performance than any one application alone. In practice, such mixed models should be expected, rather than per-application models.

As an example of a resulting tree, Figure 7.20 contains the regression tree for Array B, with no additional pruning for readability. There are a total of 76 leaf nodes (rules) for predicting the bandwidth of Array B. When compared to the tree depths of the per-applications models, the mixed models contain a much more complex set of rules. Indeed, they attempt to learn the performance characteristics of an array, rather than just the performance of a given application. To test that such complex models do not introduce additional error, the mixed models are evaluated using the same set of testing samples used to test the per-application models.

Figure 7.21 graphs the error of the mixed models. One can see the effects of changing workload characteristics by comparing the idealized absolute model against the in-practice absolute model. This result is consistent with the per-application models just presented.

Table 7.8 compares the median relative errors directly. As shown in the table, the per-application and mixed models are within 3% of one another for all workloads, with the exception of FitnessCache (a 9% difference). In some cases, prediction error increases slightly when using a mixed model, and in others it actually improves. This is because additional training data, even though it may come from different workloads, can provide useful information to CART when learning the performance characteristics of an array.

7.6.4 Summary

The amount by which changing workload characteristics increases the prediction error of an absolute model was measured across a variety of workloads and disk arrays. Overall, predictions are more accurate when the workload characteristics are obtained from the same disk array for which the prediction is being made. In support of Hypothesis 2, Experiment 2 is summarized as follows:

Result 2 *Changing workload characteristics can increase the prediction error of an absolute model. When compared to an idealized absolute model ($i = j$), an in-practice absolute model ($i \neq j$) can increase the median relative error by over a factor of two (the* FitnessDirect *latency error increases from*

34% for the idealized array pairing $D \rightarrow D$ to 77% for the pairing $A \rightarrow D$; and the `TPC-C` *latency error increases from 11% for $C \rightarrow C$ to 46% for $B \rightarrow C$). Over all array pairings, the bandwidth prediction error increase up to 3% (*`FitnessDirect` *increases from 21% to 24%), throughput up to 3% (*`FitnessDirect` *increases from 23% to 26%), and latency up to 12% (*`TPC-C` *increases from 11% to 23%).*

In general, the bandwidth and throughput predictions are not affected by changing workloads as much as the latency predictions are. This is due to the fact that their best predictors (read characteristics) are, as shown in Experiment 1, the least likely to change across arrays. For latency, however, the write queue depth can be a leading source of information, and this workload characteristic experiences significant change across arrays. It is also interesting to note that the write randomness, although it experiences significant change across arrays, is not a leading source of information. This is an example of a workload characteristic that changes but has little effect on prediction accuracy.

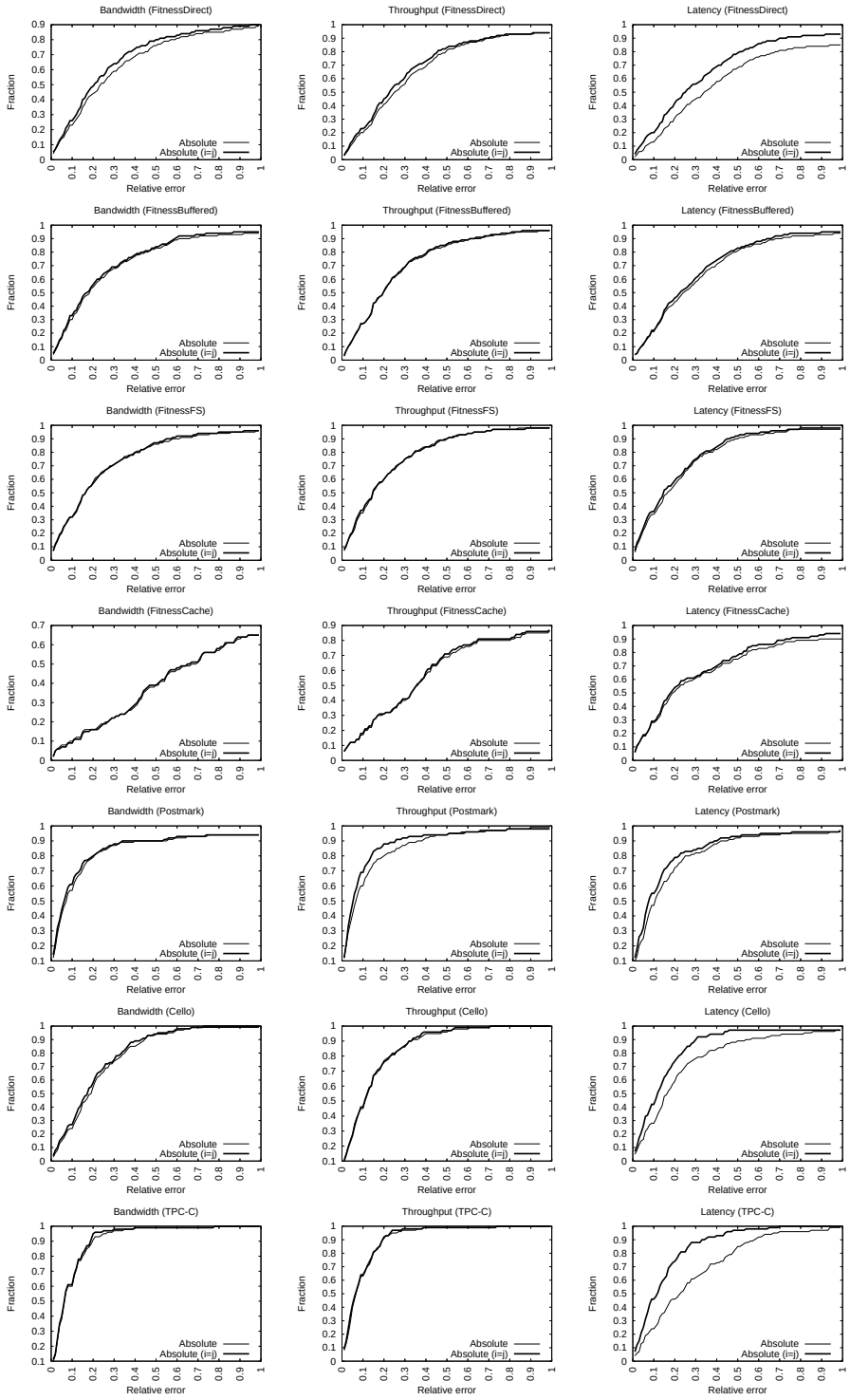

Figure 7.16: Relative error CDFs .

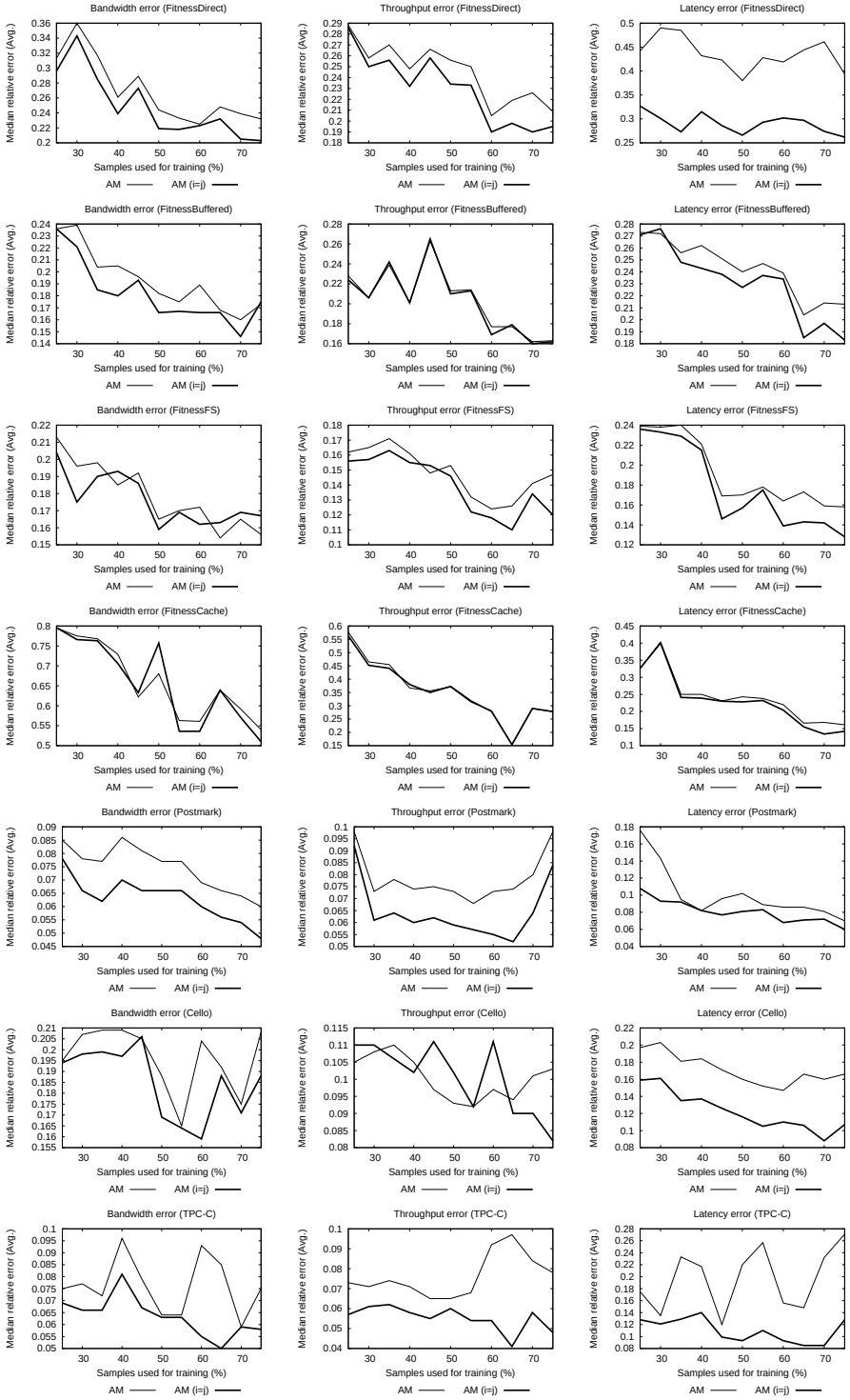

Figure 7.17: Error vs. training set size. The training set size varies from 25% of the collected samples to 75%; the remaining samples are used for testing. For each training set size, an error metric (median relative error) is calculated for each pairing of arrays. These metrics are then averaged across all pairings to produce the values shown.

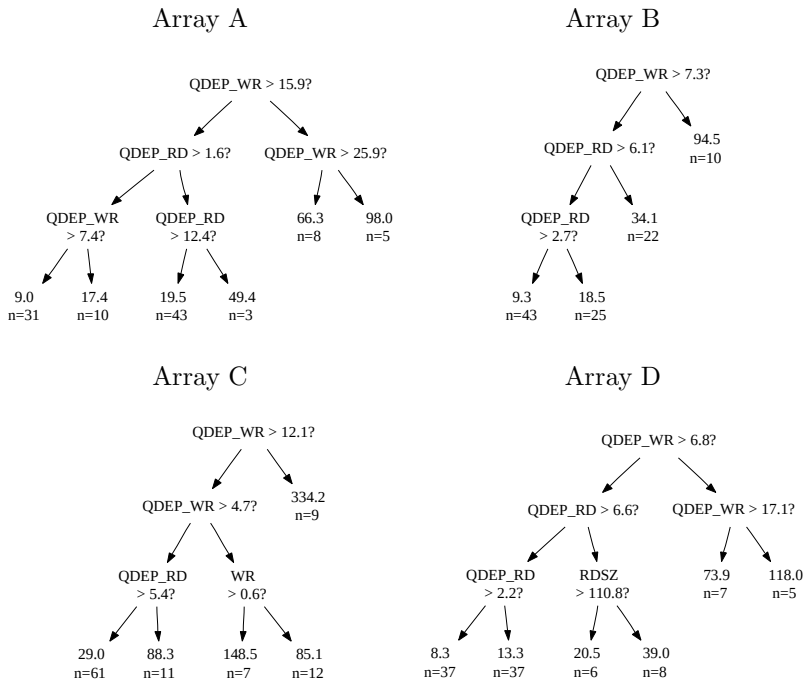

Figure 7.18: Absolute latency models for `FitnessDirect`.

.

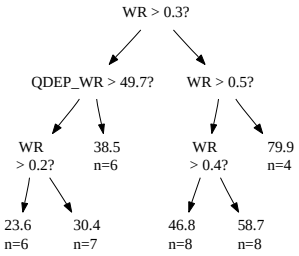

Figure 7.19: Absolute latency model of Array B for `Cello`.

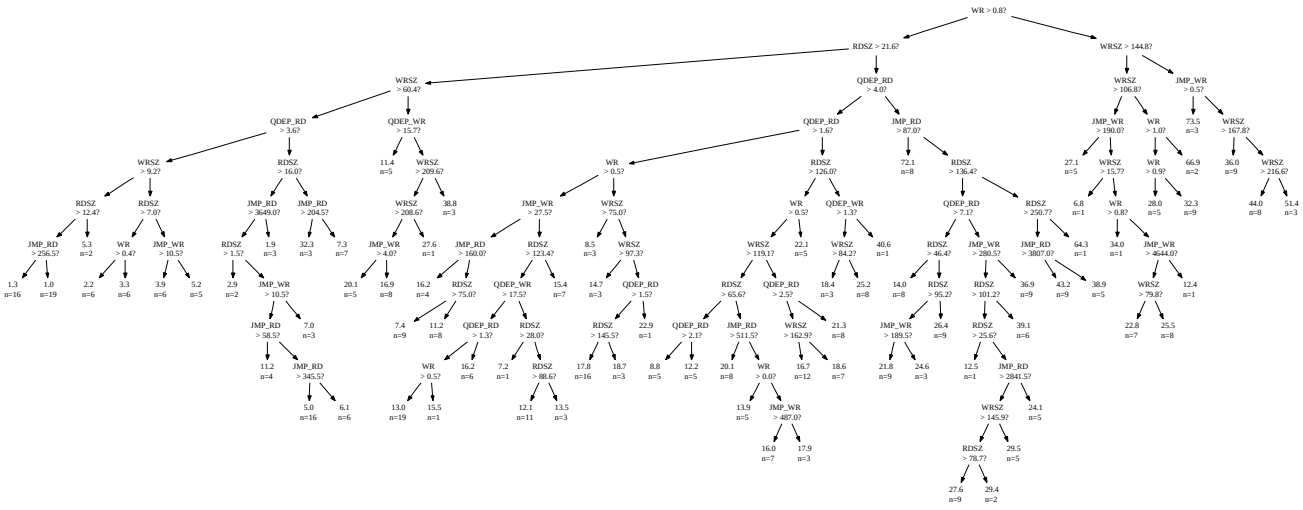

Figure 7.20: Absolute bandwidth model of Array B (`WorkloadMix`).

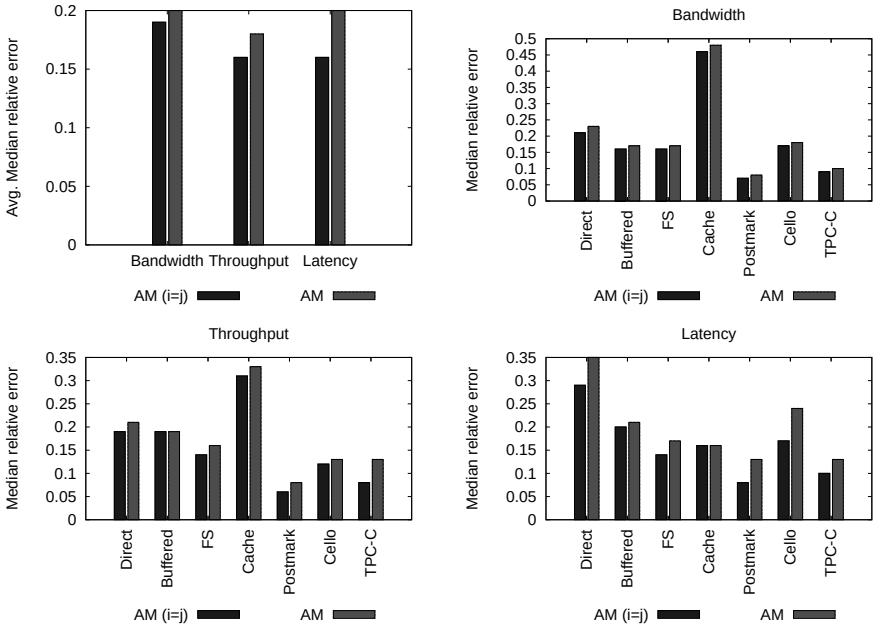

Figure 7.21: Median relative error: Experiment 2 (mixed models). The graphs show the median relative error of each application, comparing the idealized absolute model, title "AM (i=j)," with the absolute model (AM). Also shown are the average median relative errors of the applications for each performance metric.

7.7 Experiment 3: relative models

Experiment 1 confirmed that performance differences among the disk arrays can lead to changes in workload characteristics, and Experiment 2 confirmed that changes in workload characteristics can lead to increases in the prediction error of an absolute model. Hypothesis 3 asserts that one can reduce this error by modeling an array i relative to the workload characteristics of another array j.

To test Hypothesis 3, relative models are built for each array pairing and performance metric. Each model implements Equation 6.1, where only the workload characteristics of array j are used to predict performance of array i. There are 16 pairings and 3 metrics, for a total of 48 relative models. Although a relative model where $i = j$ is simply an absolute model, these models are presented separately in many of the appendix tables for improved readability. It also serves as sanity-check that the relative models and absolute models have trained and tested over the same set of samples (i.e., the prediction errors for the absolute and relative model should be identical when $i = j$).

As described in Chapter 6, the only differences between an absolute and relative model are the workload characteristics used during training. When learning the performance of array i, an absolute model is trained with workload characteristics as measured by array i, and a relative model is trained with those measured by a different array j. Summarizing from Section 7.4.3 (Table 7.2), these characteristics include the write fraction, write request size, read request size, write randomness, read randomness, write queue depth, and read queue depth. The predicted variable is the same for the absolute and relative model (i.e., the bandwidth, throughput, or latency of array i).

Just as in Experiment 2, 50% of the workload samples are used for model testing. This involves inputing the workload characteristics of the sample as measured on array j into the relative performance models of array i, for all pairings $j \rightarrow i$. The prediction accuracies are then compared against the idealized and in-practice absolute models presented in the previous section. The goal of the relative model is to reduce the prediction error of the in-practice absolute model or, stated differently, to match the prediction error of the idealized absolute model. If there is no change in workload characteristics for a pairing $j \rightarrow i$, then a relative model is, in effect, equivalent to an absolute model (same input and same output).

7.7.1 Per-application models

Relative models can reduce the error caused by changing workload characteristics.

`FitnessDirect`

Figure 7.22 contains the relative latency models of Array A for `FitnessDirect`. Also shown is the absolute latency model for Array D. In taking a closer looking at the relative performance model $D \rightarrow A$, one can see many similarities in the internal node structure with the absolute model of Array D. Again, the goal of the relative model is to learn the performance of Array A relative to the workload characteristics as measured by Array D. Therefore, it may be the case that the relative and absolute models choose the same characteristic and in some cases the same splitting value.

For example, the absolute latency model of Array D and the relative model $D \rightarrow A$ both proceed down the right half of the tree, if the write queue depth (as measured by Array D) is greater than 6.8. Naturally, if the models use the same workload characteristics (those of Array D in this case) to separate

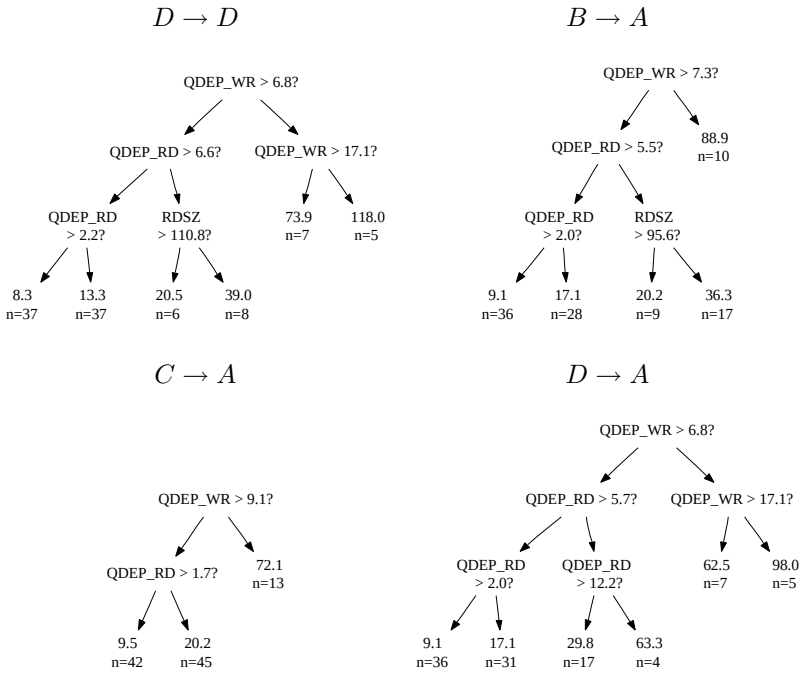

Figure 7.22: Relative latency models for `FitnessDirect`.

training samples into leaf nodes, then similar (or identical) values can be chosen for the splits. However, because a regression tree is constructed such that the error in the leaf nodes is minimized, different splitting values may be chosen, or different splits altogether, as the leaf nodes contain different values (e.g., the leaf nodes of the $D \rightarrow A$ model contain performance values for Array A and those for $D \rightarrow D$ contain performance values for Array D).

These models, illustrated, further clarify why changes in workload characteristics are a non-issue for a relative model. Because a relative model of array i is trained using the workload characteristics observed by array j, an incorrect path cannot be taken in the tree due to changes in workload characteristics between arrays j and i — because the workload characteristics of array i are not even used by the relative model.

Figure 7.23 plots the median relative error of the relative models for `FitnessDirect`. Also shown are the median relative errors for the idealized and in-practice absolute models from Experiment 2. In general, one first sees error increasing (due to changing workload characteristics) then decreasing (due to the use of a relative model). For `FitnessDirect`, the largest reduction in error is for the latency predictions, from 34% to 25%; the same increasing/decreasing effect can be seen for bandwidth and throughput, although not as pronounced.

Figure 7.24 (top row) plots the cumulative distribution functions of relative error. These are the same CDF graphs as in Experiment 2, but with the relative models added for comparison. In the latency graph, one can see that the distribution of error for the relative model (now the darkest line) is nearly identical to that of the idealized absolute model. The same is true for the bandwidth and throughput CDFs, though more difficult to see.

As with Experiment 2, the differences among the models are more easily seen in the error-versus-

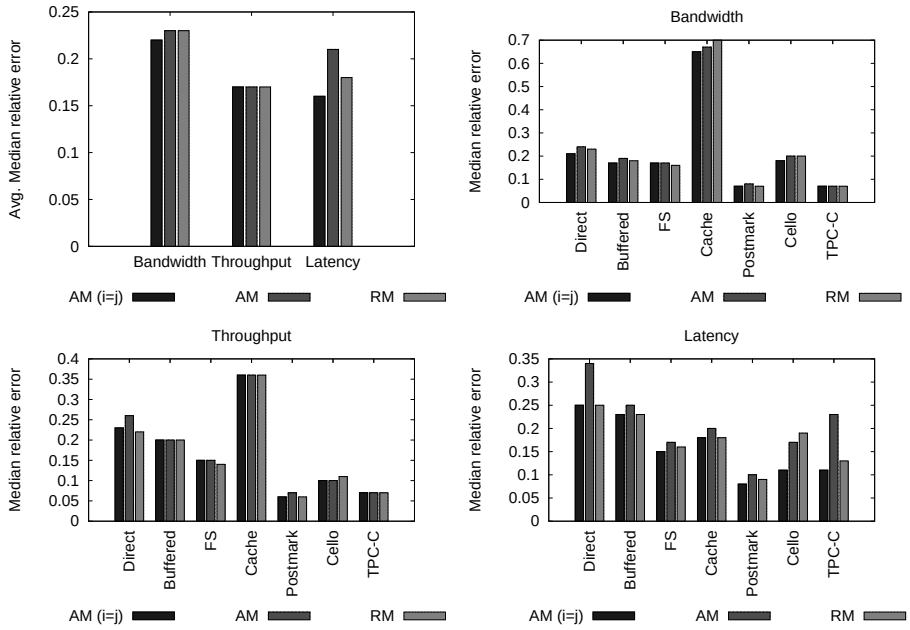

Figure 7.23: Median relative error (per-application models). The graphs show the median relative error of each application, comparing the idealized absolute model, title "AM (i=j)," with the absolute model (AM) and the relative model (RM). Also shown are the average median relative errors of the applications for each performance metric.

training graphs. Figure 7.25 (top row) plots the same graphs as in Experiment 2, with the relative models (darkest line) added for comparison. As can be seen, the relative model and idealized absolute model track each other well. This is especially evident in the latency graph.

Of course, the improvement to be had by using a relative model (one that only uses workload characteristics) is limited by the extent to which workload characteristics change between two arrays. Therefore, the predictions that suffered the most in Experiment 2, due to changing workloads, will be those that stand to benefit the most from the use of a relative model. In the case of `FitnessDirect`, these are the latency predictions. As an example, recall from the previous section that the in-practice absolute models experience the greatest error for Array A, either when using the characteristics of Array A to predict the performance of another array or when predicting the performance of Array A using the characteristics of one of the other arrays.

Specifically, when compared to the median relative error of the idealized absolute model of Array A (18%), the in-practice predictions increase error substantially: $B \rightarrow A$ is 27%, $C \rightarrow A$ is 30%, and $D \rightarrow A$ is 27%. Things are even worse when using characteristics from Array A to predict for the other arrays: $A \rightarrow B$ is 44%, $A \rightarrow C$ is 58% and $A \rightarrow D$ is 77%. Collectively, the average median relative error of the latency predictions involving Array A is 44%. However, when relative models are used to make these predictions, the average median relative error is reduced to 22%. Therefore, although the median relative error over all pairings is reduced from 34% to 25% (as shown in Figure 7.23), when looking at individual pairings, the differences can be even greater.

FitnessBuffered

As shown in Experiment 2, the FitnessBuffered predictions are only affected slightly by changing workload characteristics. Therefore, the cumulative distribution function of the idealized absolute models, in-practice absolute models and relative models are, for the most part, indistinguishable, as shown in Figure 7.24. Still, one can see differences in the error-versus-training graphs in Figure 7.25. In most of the training/testing splits, the idealized absolute models and relative models perform similarly and are more accurate than the in-practice absolute models.

Just as with FitnessDirect, one sees a literal reversing of the effects of changing workloads. For bandwidth, the median relative error increases from 17% to 19% due to changing workloads, and then decreases to 18% when using a relative model. For throughput, the median relative error of all three models is the same (20%). For the latency predictions, the median relative error of the idealized absolute models is 23%, increasing to 25% for the in-practice models, and decreasing back to 23% for the relative models.

FitnessFS

Similarly to FitnessBuffered, one sees only slight differences across the models for FitnessFS. The median relative error of the idealized absolute models, in-practice absolute models, and relative models are within 2% of one another. Again, this result is consistent with that of Experiment 2, as the best predictors for this workload exhibit the least change across disk arrays. As such, only minor improvements are to be gained by using a relative model that only uses workload characteristics. As can be seen in the cumulative distribution functions in Figure 7.24, the models have nearly identical distributions.

FitnessCache

The effect of the relative models on the FitnessCache predictions is similarly small. Prediction error increases by less than 3% due to changing workloads. For all performance metrics, the difference in median relative error among the idealized absolute models, in-practice absolute models, and relative models is within 5%. As further illustrated in the cumulative distributions functions in Figure 7.24 and the error-versus-training graphs in Figure 7.25, there is very little difference among the models.

Postmark

Although the differences are small, as shown in Figure 7.23, Postmark is another good example of a case where the relative model eliminates the error due to changing workloads characteristics, albeit only 1%, and produces the same accuracy as the idealized absolute model. Such is the case for the bandwidth and throughput predictions. The significance is not so much the reduction in error, but rather that the relative model is not expected to exceed, but simply match the accuracy of the idealized absolute model.

Further, Postmark illustrates a case where different training/testing splits result in greater differences among the models, as can be seen in the error-versus-training graphs in Figure 7.25. In particular, the average median relative error of the in-practice latency models is approximately 18% when training on 25% of the samples and testing on the remaining 75%, versus an error of 11% for the idealized and relative models.

`Cello`

Recall from Experiment 2 that the `Cello` latency predictions are affected by changing workloads (median relative error increases from 11% to 17%). Given this, one would expect a commensurate decrease in error when using a relative model. Unfortunately, this is not the case with `Cello`.

`Cello` represents a workload where a relative model cannot cope with the changing workloads and match the accuracy of the idealized absolute model. As can be seen in the cumulative distribution of latency error in Figure 7.24, the distribution of error for the relative models is more similar to that of the in-practice absolute models. The same is true for the error-versus-training graph in Figure 7.25.

Fortunately, a relative model has the option of using potentially more valuable (and less volatile) performance characteristics in place of workload characteristics. As will be shown in Experiment 4, it is not until performance is added to the relative models of `Cello` that one can exceed the accuracy of the in-practice absolute models.

`TPC-C`

Recall from Experiment 2 that the `TPC-C` latency predictions incur the largest error due to changing workloads. In particular, changes in the read/write ratio and write queue depth lead to a 12% increase in median relative error, from 11% to 23%. The relative model is able to eliminate most of this error.

As shown in Figure 7.23, the relative models reduce error by 10%, thereby yielding an accuracy within 2% of the idealized absolute model. This effect is illustrated in the cumulative distribution of error in Figure 7.24, as well as the error-versus-training graphs in Figure 7.25.

Like `FitnessDirect`, the improvements in relative error for `TPC-C` are more pronounced for specific array pairings. For example, the median relative error of the $B \rightarrow C$ latency predictions, using an absolute model, is 46%, compared to 11% for $C \rightarrow C$. However, when using a relative model, the error of $B \rightarrow C$ is reduced to 17%. The Array D latency predictions also suffer considerably from changing workloads: $D \rightarrow D$ is 12%, but $C \rightarrow D$ is 25%, $B \rightarrow D$ is 36% and $A \rightarrow D$ is 35%. Collectively, their average median relative error is 32%. However, relative models reduce this average to 11%.

7.7.2 Best predictors

Table 7.9 contains the best predictors of performance for the relative models. They are primarily the same as those of the absolute models (Table 7.7); the importance measure of each workload characteristic may vary slightly. In particular, the best predictors of bandwidth and throughput are the read characteristics (queue depth and randomness), and the best predictors of latency are the read and write queue depths.

7.7.3 Mixed models

The average median relative errors of the mixed models are within 3% of the per-application models. As can be seen in Figure 7.26, the relative models, overall, produce the lowest average median relative error for the throughput and latency predictions; the absolute models and relative models tie for the bandwidth predictions. In 12 of the 21 cases (i.e., there are 7 bandwidth graphs, 7 throughput graphs, and 7 latency graphs in Figure 7.26), the relative models reduce error over the absolute models, by up to 5%. In 8 of the cases, the absolute models produce less error, but only up to 3%. In one case (`FitnessFS`

Overall		Bandwidth		Throughput		Latency	
Predictor	**Score**	**Predictor**	**Score**	**Predictor**	**Score**	**Predictor**	**Score**
Read queue	1.00	Read queue	1.00	Read queue	1.00	Write fraction	1.00
Write fraction	0.65	Read jump	0.72	Read jump	0.67	Read queue	0.88
Read jump	0.57	Read size	0.52	Write size	0.47	Write queue	0.48
Read size	0.42	Write fraction	0.52	Read size	0.46	Read jump	0.32
Write size	0.39	Write size	0.48	Write queue	0.28	Read size	0.28
Write queue	0.37	Write jump	0.34	Write fraction	0.22	Write size	0.23
Write jump	0.24	Write queue	0.27	Write jump	0.20	Write jump	0.16

Table 7.9: The normalized importance measure of each predictor for the relative models.

latency), they tie. This result confirms that the benefits of relative models can also be seen with models that are trained with a mix of different workloads.

7.7.4 Summary

The amount by which relative models reduce the prediction error caused by changing workloads was measured across a variety of workloads and disk arrays. Overall, predictions are more accurate when the performance of array i (the workload target) is modeled relative to the workload characteristics of array j (the workload origin). In support of Hypothesis 3, Experiment 3 is summarized as follows:

Result 3 *When compared to an absolute model, a relative model can decrease the median relative error by over half (the* `FitnessDirect` *latency prediction error decreases from 77% to 29% for the pairing $A \rightarrow D$, and* `TPC-C` *decreases from 46% to 17% for $B \rightarrow C$). Over all pairings, the bandwidth prediction error decreases up to 1% (*`FitnessDirect` *decreases from 24% to 23%), throughput up to 4% (*`FitnessDirect` *decreases from 26% to 22%), and latency up to 10% (*`TPC-C` *decreases from 23% to 13%).*

It is no coincidence that the above result is almost an exact complement of Experiment 2. Recall that in Experiment 2, `FitnessDirect` and `TPC-C` are the workloads where prediction accuracy suffers the most from changing workloads. They are also the workloads that benefit the most from a relative model.

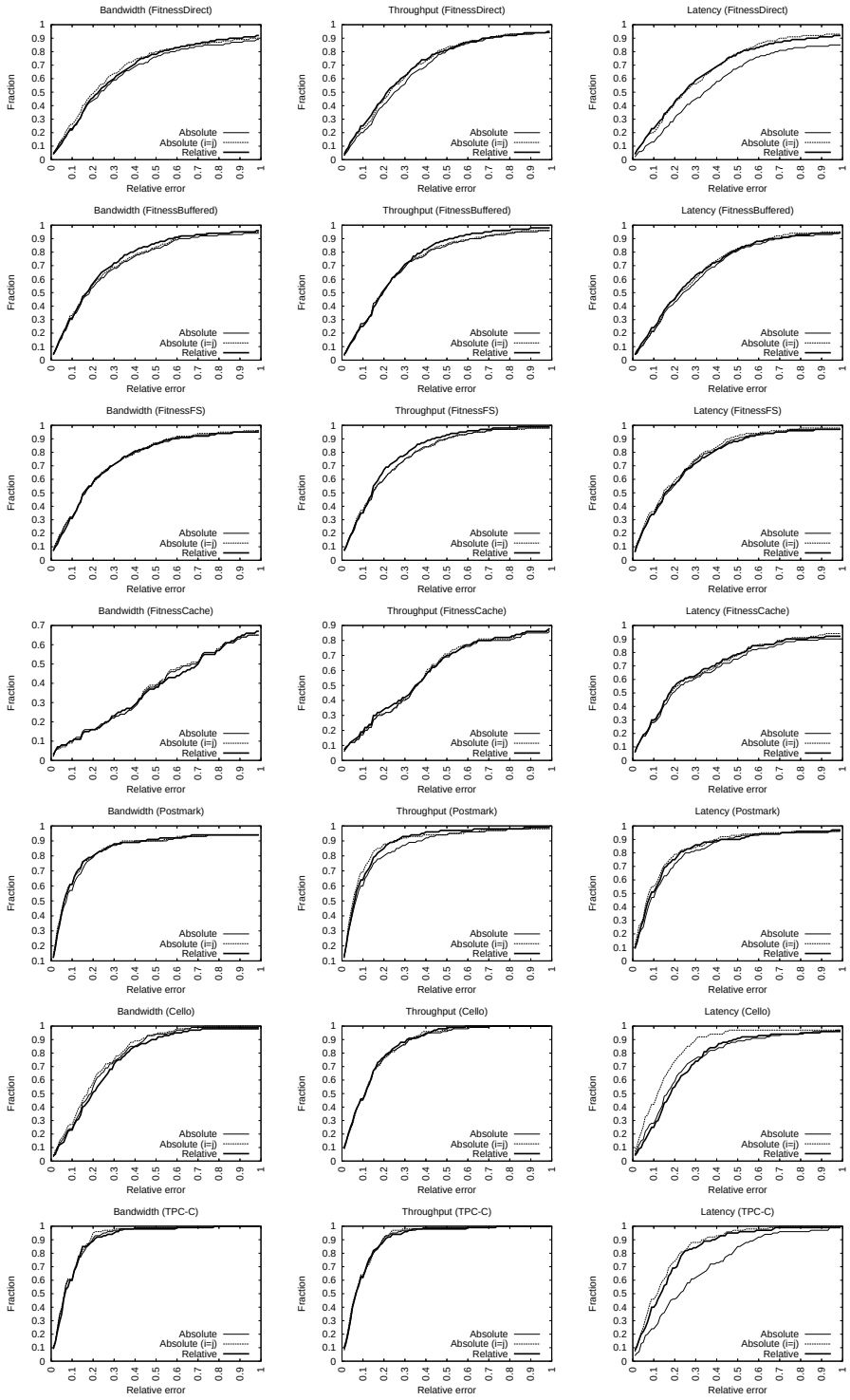

Figure 7.24: Relative error CDFs.

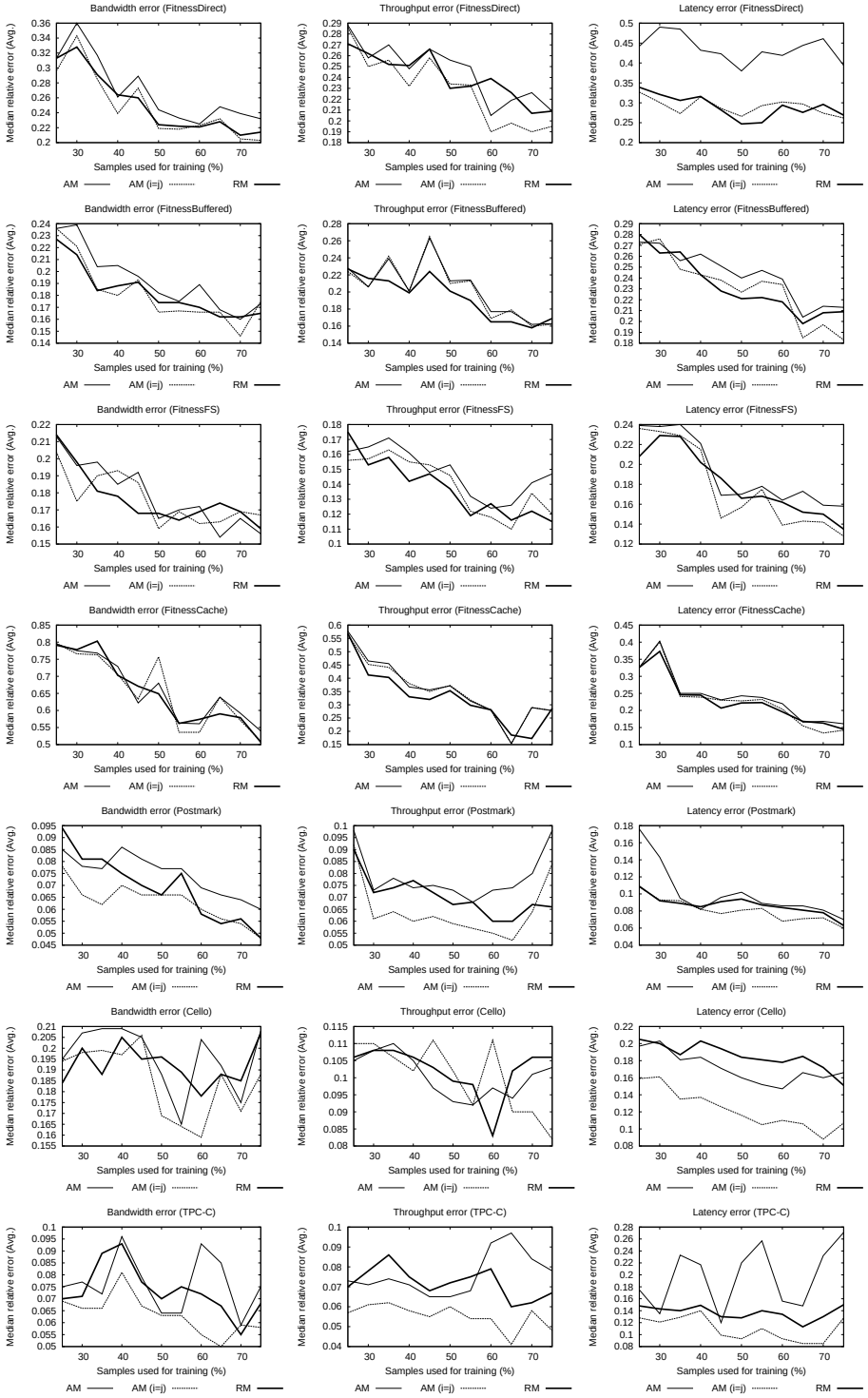

Figure 7.25: Error vs. training set size. The training set size varies from 25% of the collected samples to 75%; the remaining samples are used for testing. For each training set size, an error metric (median relative error) is calculated for each pairing of arrays. These metrics are then averaged across all pairings to produce the values shown.

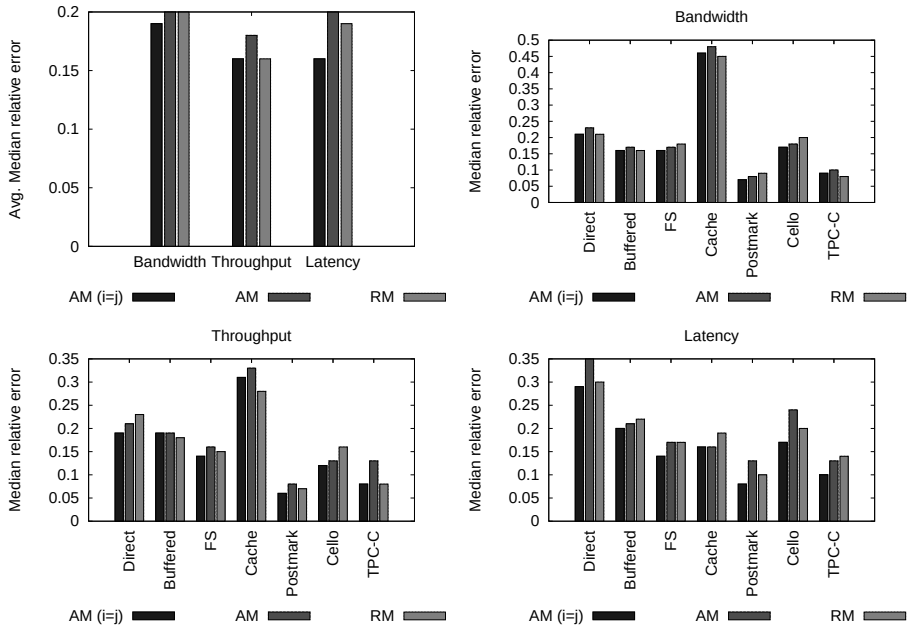

Figure 7.26: Median relative error (mixed models). The graphs show the median relative error of each application, comparing the idealized absolute model, titled "AM (i=j)," with the absolute model (AM), and the relative model (RM). Also shown are the average median relative errors of the applications for each performance metric.

7.8 Experiment 4: relative performance models

Experiments 1 through 3 established that changing workload characteristics can increase the prediction error of an absolute model and that this error can often be reduced, or eliminated, by using a relative model. The goal of Experiment 4 is to extend a relative model with performance observations, thereby creating a relative performance model (Equation 6.2). As described in Section 6.3, a relative performance model uses the observed workload characteristics and performance metrics from array j to predict the performance of array i.

The test of Hypothesis 4 is similar to that of Hypothesis 3: two relative models ($i \rightarrow j$ and $j \rightarrow i$) are built for each array pairing and performance metric. Summarizing from Section 7.4.3 (Table 7.2), the workload characteristics are the same as those of used by the absolute and relative models, including the write fraction, write request size, read request size, write randomness, read randomness, write queue depth, and read queue depth. The additional predictor variables (Table 7.2) include the bandwidth, throughput, and latency of array j, with latency further described by the write latency and read latency.

Just as in Experiments 2 and 3, each of the workload samples reserved for testing is used to measure the accuracy of the relative performance models. For each performance metric, two predictions are made for every array pairing ($i \rightarrow j$ and $j \rightarrow i$). This involves inputing the workload characteristics and performance of the sample as measured on array j into the relative performance model for array j to i, and vice versa. The prediction errors are then compared against the relative models to see the incremental benefit of using the observed performance of one array to predict the performance of another.

7.8.1 Per-application models

Observed performance is a good predictor of future performance.

`FitnessDirect`

The best predictor of performance for `FitnessDirect` is, in fact, the observed performance of another array. Figure 7.27 contains the relative performance models for latency. Also shown is the absolute model of Array A from Experiment 3. One notices that the latency of array j is the strongest predictor of the latency of array i. Intuitively, this stands to reason. Rather than attempt to learn how queue depths, request sizes, and spatial randomness influence I/O latency, a model can simply observe the latency of array j and translate this into a latency of another array i.

Table 7.10 shows the normalized importance measure for each attribute, across all array pairings. In general, the best predictor of the bandwidth of array i is the bandwidth of array j. Similarly, the best predictor of throughput is throughput, and the best predictor of latency is latency. Also note that the second and third best predictors are often performance metrics. In other words, the natural (mathematical) connection among bandwidth, throughput and latency is automatically discovered by CART. However, workload characteristics (e.g., read queue depth) are still found to be valuable. So, rather than discard them, the best solution is to allow a model to determine which combinations of performance and workload characteristics for array j are most correlated with the performance of array i.

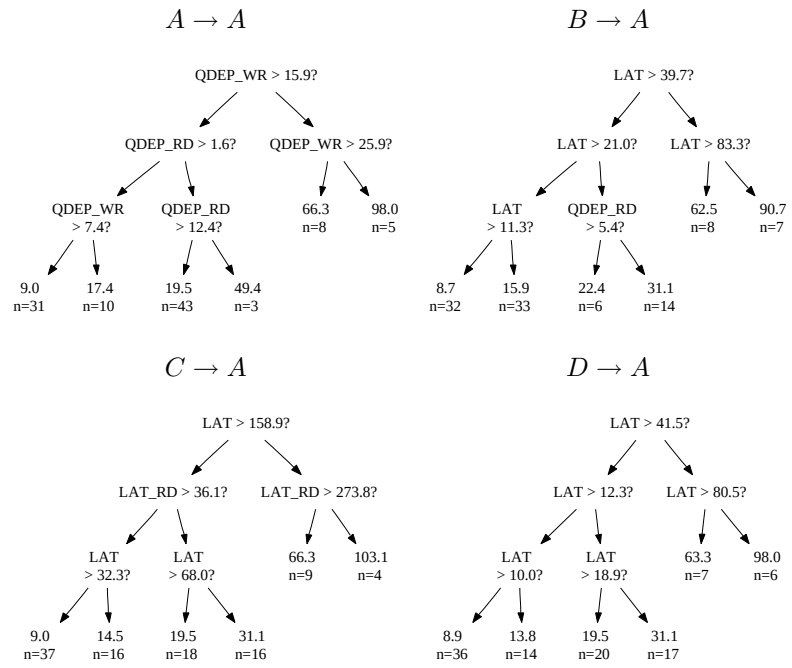

Figure 7.27: Relative performance models of the latency of Array A (`FitnessDirect`).

.

These strong correlations with performance can lead to lower prediction error. When comparing the relative performance models to the relative models, one sees further reductions in error. Figure 7.28 plots the median relative error for all applications. One can see that, in nearly all cases, the relative performance models are more accurate than the relative models. For `FitnessDirect`, the median relative error of the bandwidth predictions is reduced from 23% to 19% and the latency predictions from 25% to 20%; throughput remains the same (22%). These differences can be seen in the cumulative error distributions in Figure 7.29 as well as the error-versus-training graphs in Figure 7.30. In these graphs, the darkest line is that of the relative performance model.

`FitnessBuffered`

The improvements for `FitnessBuffered` are even more pronounced than those of `FitnessDirect`. The median relative error of the bandwidth predictions decreases from 18% to 14%, throughput from 20% to 15%, and latency from 23% to 16%. These differences can be seen in the cumulative error distributions in Figure 7.29 as well as the error-versus-training graphs in Figure 7.30. Of all the applications, the `FitnessBuffered` graphs are the most salient in terms of error reduction from the relative performance models.

The strongest (top 3) bandwidth predictors for `FitnessBuffered` are the bandwidth, followed by the read randomness, and then the read latency. For throughput, they are the throughput, the read queue depth, and the read randomness. For latency, they are the latency, the read/write ratio and the bandwidth. Unlike `FitnessDirect`, the models of `FitnessBuffered` make greater use of the workload characteristics. Still, the strongest predictors are the performance metrics.

Relative performance model							
Overall		**Bandwidth**		**Throughput**		**Latency**	
Predictor	**Score**	**Predictor**	**Score**	**Predictor**	**Score**	**Predictor**	**Score**
Latency	1.00	Bandwidth	1.00	Throughput	1.00	Latency	1.00
Bandwidth	0.83	Latency	0.24	Bandwidth	0.26	Write latency	0.16
Throughput	0.59	Write fraction	0.15	Latency	0.11	Read latency	0.16
Read queue	0.19	Write jump	0.12	Read queue	0.11	Bandwidth	0.10
Read latency	0.17	Throughput	0.11	Write queue	0.08	Read queue	0.09
Write fraction	0.17	Read size	0.10	Write size	0.05	Write queue	0.08
Write latency	0.16	Read queue	0.10	Write fraction	0.05	Read jump	0.08
Write queue	0.16	Read jump	0.09	Read size	0.05	Write fraction	0.06
Read jump	0.14	Write queue	0.09	Write jump	0.04	Read size	0.05
Read size	0.13	Read latency	0.08	Read jump	0.04	Write size	0.02
Write jump	0.11	Write size	0.05	Write latency	0.02	Write jump	0.02
Write size	0.07	Write latency	0.04	Read latency	0.01	Throughput	0.01

Table 7.10: The normalized importance measure of each predictor.

`FitnessFS`

One can see similarly large reductions in median relative error for `FitnessFS`. When compared to the relative model, the relative performance model reduces bandwidth prediction error from 16% to 11%, throughput from 14% to 11%, and latency from 16% to 14%. As with `FitnessDirect` and `FitnessBuffered`, the distributions of error for the relative performance models are distinguished from the relative and absolute models in Figure 7.29. The same is true in the error-versus-training graphs in Figure 7.30.

The top predictors of performance for `FitnessFS` are the performance metrics (bandwidth is the best predictor of bandwidth, etc.). However, the second and third best predictors are workload characteristics. For bandwidth, they are the read queue depth and read randomness. For throughput, they are the write size and read queue depth. For latency, they are the read/write ratio and the read randomness. In this regard, `FitnessFS` is a good example of how useful workload characteristics are, even when the top predictors are performance metrics.

`FitnessCache`

The `FitnessCache` workload exhibits the largest reduction in median relative error, particularly for the bandwidth predictions, where error is reduced from 70% to 47%. Throughput experiences no change (36%) and latency only a small reduction, from 18% to 17%. One can see the reduction in bandwidth error in the error distributions graphs in Figure 7.29 and in the error-versus-training graphs in Figure 7.30.

Upon a closer inspection of the regression trees, it is found that the relative performance models of bandwidth are able to create more homogeneous leaf nodes, by splitting on the bandwidth metric. For example, the $C \rightarrow A$ relative performance model predicts 71.3 MB/sec for `FitnessCache` sample #64, and the leaf used to make this prediction has a mean absolute deviation of 9.4 MB/sec. That is, the training samples mapping to the leaf node have a median value of 71.3 (the prediction) with a mean absolute deviation of 9.4 from their average value. The actual performance of sample #64 on Array A is 79.3 MB/sec, for a relative error of 10%.

In contrast, the relative models that only use workload characteristics can have less homogeneous

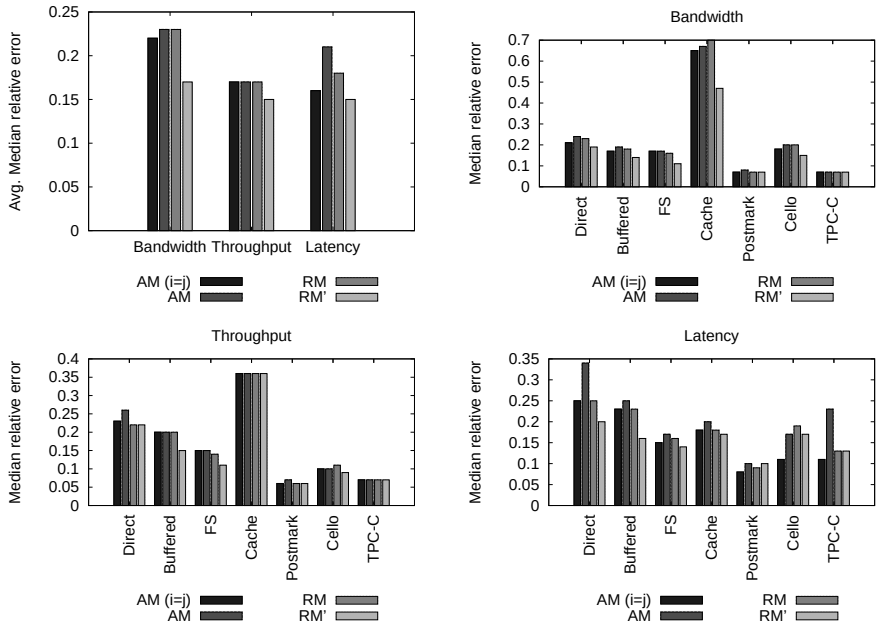

Figure 7.28: Median relative error: Experiment 4 (per-application models). The graphs show the median relative error of each application, comparing the idealized absolute model, title "AM (i=j)," with the absolute model (AM), the relative model (RM), and the relative performance model (RM'). Also shown are the average median relative errors of the applications for each performance metric.

leaf nodes. For sample #64, the relative bandwidth model for $C \rightarrow A$ predicts 46.9 MB/sec, with a mean absolute deviation of 17.0 MB/sec. The relative error of the prediction is 41%. Had the relative model had additional workload characteristics to better separate the samples, it may have created a more homogeneous leaf node, resulting in a more accurate prediction. In effect, a performance characteristic is really just a different type of workload characteristic, just one that is specific to a given array.

Also, it is interesting to note that the relative performance models can trivially distinguish the `FitnessCache` workloads that have a high cache hit rate from those that do not. The only workload characteristic that somewhat reflects this behavior is the read randomness. However, what is really needed is a more direct measure of locality, and this is exactly what the performance metrics provide. For `FitnessCache`, high bandwidth equates to "in cache" and low bandwidth to "out of cache." As such, even without a direct measure of spatial locality, the relative performance model can easily separate the workloads with different cache behavior.

`Postmark`

The median relative error of the `Postmark` predictions is already very small, even for the in-practice absolute models. As such, further reduction in relative error is going to be difficult. In the case of the relative performance models, bandwidth remains unchanged (7%) as does throughput (6%). The latency error increases slightly from 9% to 10%. The similarity of all the performance models can be seen in the cumulative distribution functions (Figure 7.29) and error-versus-training graphs (Figure 7.30).

`Cello`

`Cello` is a bit more interesting. The median relative error of the bandwidth predictions decreases from 20% to 15%, throughput from 11% to 9%, and latency from 19% to 17%. The reduction in bandwidth prediction error can be seen in the error-versus-training graphs. In particular, notice how the relative performance models for bandwidth continue to improve as the training samples are increased. When training on 75% of the samples, the relative performance models have an average median relative error less than 14%, compared to an error greater than 20% for the in-practice absolute and relative models. In this case, the performance metrics enable the relative model to learn at a faster rate than the models that only use workload characteristics. Indeed, testing error decreases for the relative performance model, as more training samples are added. For the other models, testing error increases.

`TPC-C`

Similarly to `Postmark`, the prediction error for `TPC-C` is already low for the relative model. Therefore, further improvements are limited. Bandwidth error remains the same at 7%, as does throughput. Latency error also remains unchanged at 13%. The cumulative distribution functions (Figure 7.29) and error-versus-training graphs (Figure 7.30) illustrate.

7.8.2 Best predictors

Overall, the best predictors of the relative performance models are, in fact, the observed performance of other arrays. Table 7.11 shows the normalized importance measure for each attribute. As can be seen in the table, the best overall performance predictors are the bandwidth, latency, and throughput. Specifically, bandwidth is the best predictor of bandwidth, throughput is the best predictor of throughput, and latency is the best predictor for latency.

Still, workload characteristics are found to be valuable. Overall, the read characteristics are found to have more information than the write characteristics. Given the ability of disk arrays to optimize write requests (write-back caching, request coalescing, etc.), it is intuitive that the write I/O characteristics influence performance much less than those of reads.

7.8.3 Mixed models

The accuracies of the mixed models are consistent with those of the per-application models. As can be seen in Figure 7.28, the relative performance models, overall, produce the lowest average median relative error for the bandwidth, throughput, and latency predictions. In all but one case (`TPC-C` bandwidth, which experiences a 1% increase in error), the relative performance models are as good or better than the relative models. This confirms that a single relative performance model can accurately predict the performance of a disk array when the model is trained over a variety of application types.

7.8.4 Summary

Relative performance models were constructed to measure the benefits of using performance observations to predict performance. Predictions are more accurate when the performance of array i (the workload

Overall		Bandwidth		Throughput		Latency	
Predictor	**Score**	**Predictor**	**Score**	**Predictor**	**Score**	**Predictor**	**Score**
Latency	1.00	Bandwidth	1.00	Throughput	1.00	Latency	1.00
Bandwidth	0.92	Read jump	0.16	Read queue	0.22	Write fraction	0.13
Throughput	0.79	Write fraction	0.14	Read size	0.15	Bandwidth	0.11
Read queue	0.32	Read queue	0.13	Write size	0.13	Read queue	0.09
Read jump	0.28	Latency	0.11	Read jump	0.13	Read jump	0.08
Write fraction	0.26	Write size	0.10	Bandwidth	0.11	Read latency	0.08
Read size	0.22	Read latency	0.09	Latency	0.11	Write latency	0.08
Write size	0.21	Read size	0.07	Write fraction	0.07	Read size	0.08
Read latency	0.15	Throughput	0.07	Write queue	0.05	Write queue	0.06
Write queue	0.12	Write jump	0.06	Write jump	0.03	Throughput	0.06
Write jump	0.10	Write queue	0.04	Read latency	0.02	Write size	0.05
Write latency	0.09	Write latency	0.02	Write latency	0.02	Write jump	0.04

Table 7.11: The normalized importance measure of each predictor for the relative performance models.

target) is modeled relative to the workload characteristics and the performance of a different array j (the workload origin). In support of Hypothesis 4, Experiment 4 is summarized as follows:

Result 4 *When compared to a relative model that only uses workload characteristics, a relative performance model can reduce the median relative error of the bandwidth predictions up to 23% (*`FitnessDirect` *is reduced from 70% to 47%), throughput up to 5% (*`FitnessBuffered` *is reduced from 20% to 15%), and latency up to 7% (*`FitnessBuffered` *is reduced from 23% to 16%). Moreover, it is found that the best predictors of performance are performance observations. Specifically, the best predictor of the bandwidth of array i is that of another array j; the same is true for throughput and latency.*

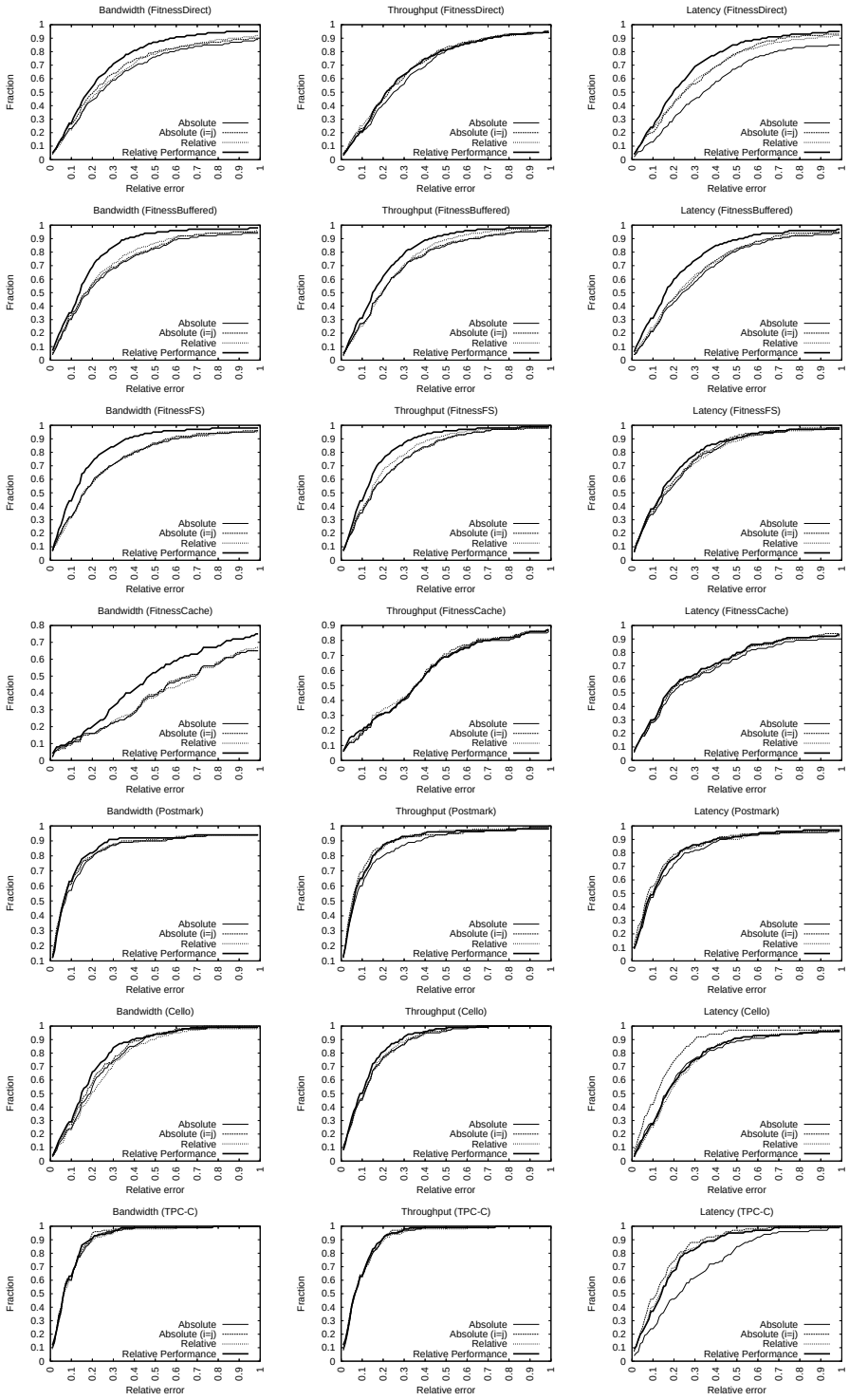

Figure 7.29: Relative error CDFs.

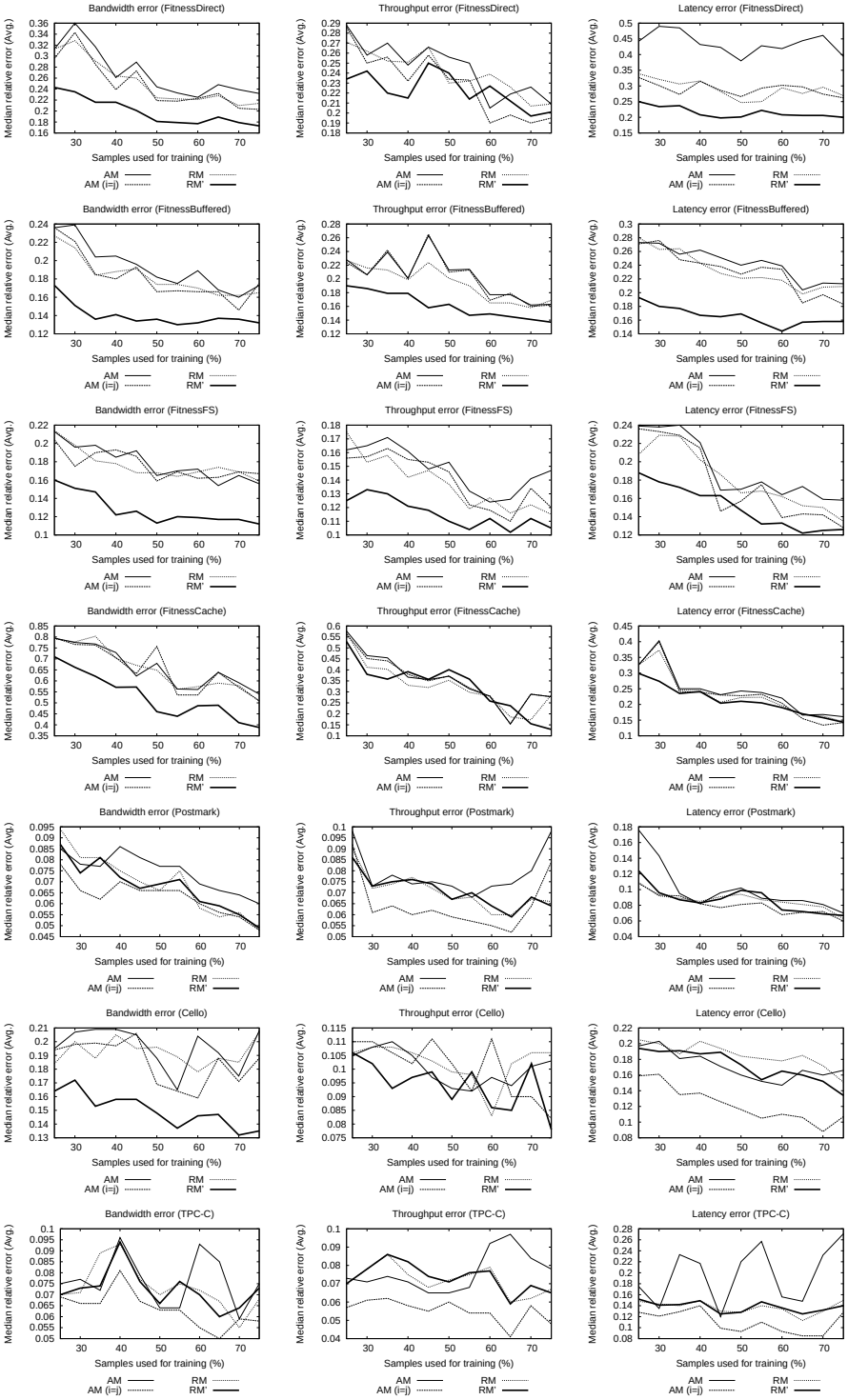

Figure 7.30: Error vs. training set size. The training set size varies from 25% of the collected samples to 75%; the remaining samples are used for testing. For each training set size, an error metric (median relative error) is calculated for each pairing of arrays. These metrics are then averaged across all pairings to produce the values shown.

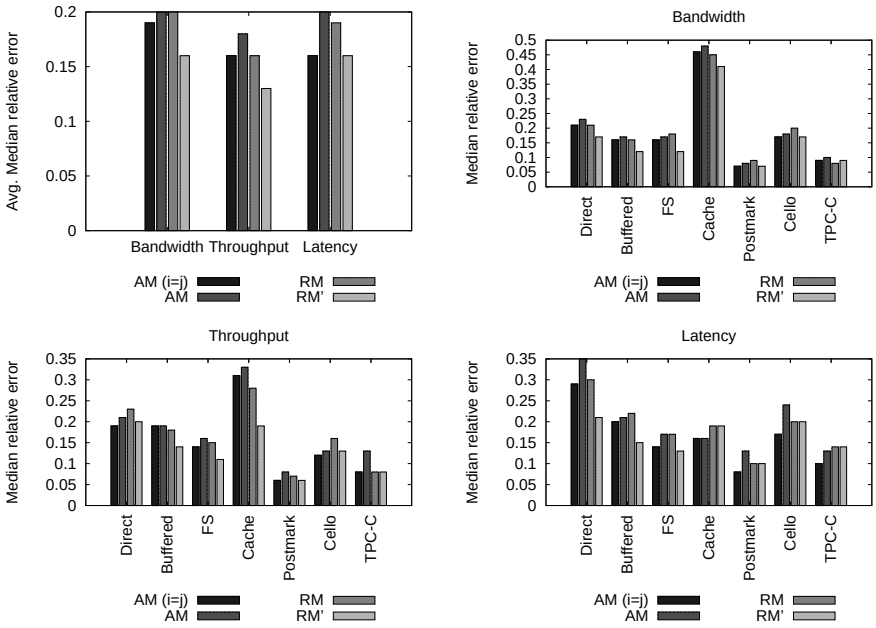

Figure 7.31: Median relative error: Experiment 4 (mixed models). The graphs show the median relative error of each application, comparing the idealized absolute model, titled "AM (i=j)," with the absolute model (AM), the relative model (RM), and the relative performance model (RM'). Also shown are the average median relative errors of the applications for each performance metric.

7.9 Experiment 5: relative fitness models

Experiments 1 through 4 established that changing workload characteristics can increase the prediction error of an absolute model, that this error can often be reduced by using a relative model, and that performance can be used to predict performance. The goal of this last experiment is to show that performance ratios, or relative fitness values, are better predictors than performance values. As described in Section 6.3, a relative fitness model uses the observed workload characteristics and performance of array j to predict a performance ratio for array i.

The test of Hypothesis 5 is the same as that for Hypotheses 3 and 4: two relative fitness models ($i \rightarrow j$ and $j \rightarrow i$) are built for each array pairing and performance metric. Summarizing from Section 7.4.3 (Table 7.2), the workload characteristics are the same as those used by the relative and relative performance models. These include the write fraction, write request size, read request size, write randomness, read randomness, write queue depth, and read queue depth. The performance metrics include the the bandwidth, throughput, and latency of array j, with latency further described by the write latency and read latency.

Just as in Experiments 3 and 4, each of the workload samples reserved for testing is used to measure the accuracy of the relative fitness models. For each performance metric, two predictions are made for every array pairing ($i \rightarrow j$ and $j \rightarrow i$) for each test sample. This involves inputing the workload characteristics and performance of the sample as measured on array j into the $j \rightarrow i$ relative fitness model, and then multiplying the predicted performance ratio by the performance of array j in order to predict the performance of array i, and vice versa.

The prediction accuracies are first compared against the relative performance models, to show the incremental benefit of using performance ratios. Then, they are compared back to the absolute models, to show the cumulative benefits of 1) relative modeling, 2) using performance to predict performance, and 3) predicting performance ratios instead of performance values.

7.9.1 Per-application models

Performance ratios are better predictors than performance values.

`FitnessDirect`

Figure 7.32 plots the median relative error for the relative fitness models. When compared to the relative performance models, the median relative error of the bandwidth predictions for `FitnessDirect` decreases from 19% to 14%, throughput from 22% to 14%, and latency from 20% to 14%. The incremental benefit of predicting the relative fitness values can be seen in the error distributions in Figure 7.33 (darkest line), as well as the error-versus training graphs in Figure 7.34. In the error-versus-training graphs, one can also see that the relative fitness predictions experience fewer fluctuations in accuracy when compared to the other models. When compared back to the absolute models, the bandwidth prediction error is reduced from 24% to 14%, throughput from 26% to 14%, and latency from 34% to 14%.

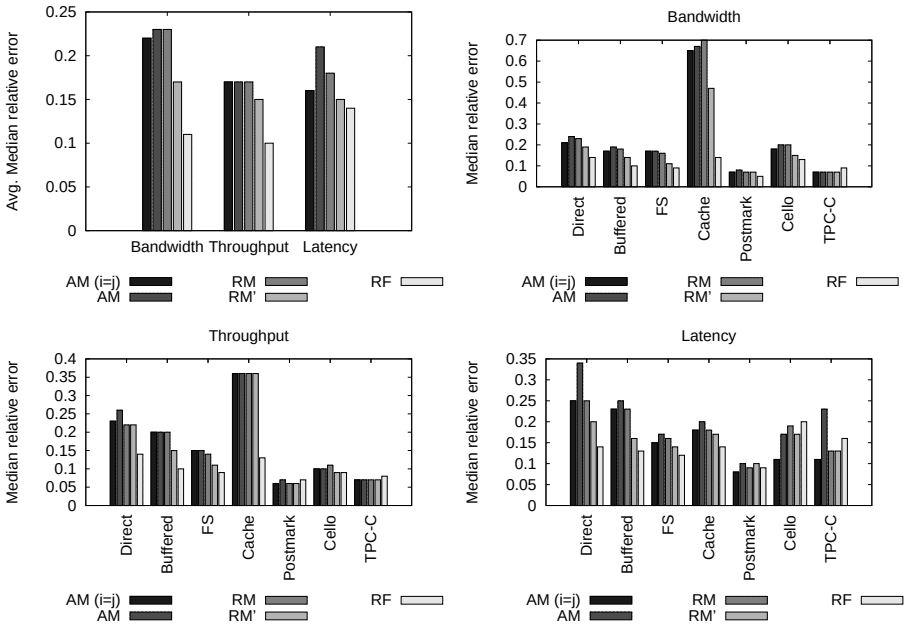

Figure 7.32: Median relative error: Experiment 5 (per-application models). The graphs show the median relative error of each application, comparing the idealized absolute model, title "AM (i=j)," with the absolute model (AM), the relative model (RM), the relative performance model (RM'), and the relative fitness model (RF). Also shown are the average median relative errors of the applications for each performance metric.

`FitnessBuffered`

When compared to the relative performance models, the bandwidth prediction error of `FitnessBuffered` decreases from 14% to 10%, throughput from 15% to 10%, and latency from 16% to 13%. When compared to the absolute models, bandwidth prediction error is reduced from 19% to 10%, throughput from 20% to 10%, and latency from 25% to 13%. That is, median relative error is reduced by roughly half.

Figure 7.33 illustrates these differences. For all performance metrics, the relative fitness models produce the least error. These differences can be seen in the error-versus-training graphs in Figure 7.34.

`FitnessFS`

When compared to the relative performance models, bandwidth and throughput prediction errors both decrease from 11% to 9%, and latency from 14% to 12%. When compared to the absolute models, bandwidth prediction error decreases from 17% to 9%, throughput from 15% to 9%, and latency from 17% to 12%.

As shown in the error distributions in Figure 7.33, the relative fitness models produce the least error. Further, across various training/testing splits, Figure 7.34 illustrates how the relative fitness models can reduce error by more than half. For example, when training on only 25% of the samples, the average median relative error of the relative fitness models is still less than 10%, compared to the absolute models with an error greater than 20%.

`FitnessCache`

The largest improvements are seen for `FitnessCache`. When compared to the relative performance models, bandwidth prediction error decreases from 47% to 14%. This represents the largest incremental reduction in error from of any of the models (over a factor of three). Further, throughput prediction error decreases from 36% to 13% (nearly a factor of three), and latency from 17% to 14%.

When compared back to the absolute models, the gains are even more pronounced. Bandwidth prediction error decreases from 67% to 14% (nearly a factor of 5), throughput from 36% to 13%, and latency from 20% to 14%.

The error distributions in Figure 7.33 and the error-versus-training graph in Figure 7.34 also show the benefits of the relative fitness models for `FitnessCache`.

`Postmark`

Despite the already low prediction error for `Postmark`, when compared to the relative performance models, the relative fitness models decrease bandwidth prediction error by 2%, from 7% to 5%. Throughput prediction error increases by 1%, from 6% to 7%, and latency decreases by 1%, from 10% to 9%.

Overall, the differences in median relative error across all the models are minor. Bandwidth predictions are within 3%, throughput predictions within 1%, and latency predictions with 1%. Nonetheless, Figure 7.33 shows that the relative fitness models improve the error distribution over the absolute models.

`Cello`

When compared to the relative performance models, bandwidth prediction error decreases from 15% to 13%, and throughput remains unchanged (9%). These small improvements can be seen in the error CDFs in Figure 7.33. Latency prediction error, however, increases by 3%, from 17% to 20%. When compared to the absolute models, bandwidth prediction error is reduced from 20% to 13% and throughput from 10% to 9%. Latency prediction error, however, still experiences the same 3% increase, from 17% to 20%.

The `Cello` latency predictions represent the only case where none of the relative models (relative, relative performance, or relative fitness) are able to achieve the accuracy of the idealized absolute model. As shown in Figure 7.33, the idealized model has the best error distribution for latency.

For bandwidth and throughput, however, one can see the improvements of relative fitness in the error-versus-training graphs in Figure 7.34. Also, the bandwidth graph in Figure 7.34 is another good example of how the error of the relative fitness models can fluctuate much less than the other models.

`TPC-C`

When compared to the relative performance models, `TPC-C` sees a slight increase in median relative error for each of the performance metrics. Bandwidth prediction error increases from 7% to 9%, throughput from 7% to 8%, and latency from 13% to 16%. As such, `TPC-C` is the only application in this evaluation where relative fitness modeling does not show a benefit over relative performance modeling, for any of the performance predictions.

When compared to the absolute models, one sees the same small increases in error for bandwidth and throughput. Latency prediction error, however, is reduced from 23% to 16%. For `TPC-C`, the only

Overall		Bandwidth		Throughput		Latency	
Predictor	**Score**	**Predictor**	**Score**	**Predictor**	**Score**	**Predictor**	**Score**
Latency	1.00	Bandwidth	1.00	Write fraction	1.00	Latency	1.00
Write fraction	0.92	Write fraction	0.99	Bandwidth	0.98	Write fraction	0.70
Bandwidth	0.89	Latency	0.91	Latency	0.92	Bandwidth	0.64
Read latency	0.61	Read jump	0.85	Read latency	0.85	Throughput	0.45
Read jump	0.61	Read queue	0.82	Read jump	0.77	Read size	0.43
Read queue	0.60	Read latency	0.74	Throughput	0.64	Write jump	0.37
Throughput	0.60	Throughput	0.66	Read queue	0.60	Read queue	0.37
Read size	0.45	Read size	0.46	Read size	0.40	Read latency	0.26
Write size	0.33	Write size	0.41	Write size	0.36	Read jump	0.24
Write queue	0.32	Write queue	0.40	Write queue	0.34	Write size	0.22
Write jump	0.28	Write latency	0.23	Write latency	0.30	Write latency	0.21
Write latency	0.25	Write jump	0.17	Write jump	0.22	Write queue	0.21

Table 7.12: The normalized importance measure of each predictor for the relative fitness models.

performance metric showing any substantial change in prediction error across all the models is latency. As shown in Figure 7.33, the error distributions of the models are nearly indistinguishable for bandwidth and throughput. But, for latency, one can see the benefits of relative fitness modeling. Further, the error-versus-training graphs in Figure 7.34 illustrate how the largest errors occur for the absolute models.

7.9.2 Best predictors

As with the relative performance models, the best predictors of the relative fitness models are the performance metrics. Table 7.12 shows the normalized importance measure for each attribute. As can be seen in the table, the best overall performance predictors are the bandwidth, latency, and throughput. The next best are the read characteristics, and the least useful are the write characteristics.

7.9.3 Mixed models

The mixed model prediction errors are within 2% of the per-application models. As shown in Figure 7.35, the relative fitness models are the best predictors, overall. In all but a few cases (`Postmark` throughput, `Cello` latency, and `TPC-C`), the relative fitness models reduce median relative error when compared to the relative performance models. In all but one case (TPC-C latency), the relative fitness models improve upon the absolute models.

Model complexity

Relative fitness models can substantially reduce prediction error, and often do so with much simpler models. Indeed, as per Occam's Razor, perhaps this is why they tend to be more accurate. Table 7.13 contains the tree complexity for all of the mixed models. Each number represents the number of leaf nodes (or prediction rules) for a particular regression tree. Averages are also shown.

The relative fitness models are generally smaller. For example, the relative fitness latency model ($B \rightarrow A$) has 20 leaf nodes, compared to 59 leaf nodes for the absolute model of Array A. In other words, the absolute model requires nearly three times as many rules to model the same disk array. As illustrated in Figure 7.36, performance ratios can be easier to model.

Model	Bandwidth	Throughput	Latency	Average
Absolute	78.8	69.2	75.5	74.5
Relative	85.6	55.2	65.5	68.8
Relative Performance	71.9	64.3	66.8	67.7
Relative Fitness	58.2	52.6	62.8	57.9

Pairwise				
Absolute	**ArrayA**	**ArrayB**	**ArrayC**	**ArrayD**
ArrayA	68.7	-	-	-
ArrayB	-	86.0	-	-
ArrayC	-	-	61.7	-
ArrayD	-	-	-	81.7
Relative Fitness	**ArrayA**	**ArrayB**	**ArrayC**	**ArrayD**
ArrayA	1	64.0	65.7	48.3
ArrayB	30.3	1	52.3	67.0
ArrayC	81.7	78.7	1	72.0
ArrayD	43.7	56.3	34.3	1

	Bandwidth				Throughput				Latency			
Absolute	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
ArrayA	83.0	-	-	-	64.0	-	-	-	59.0	-	-	-
ArrayB	-	76.0	-	-	-	88.0	-	-	-	94.0	-	-
ArrayC	-	-	66.0	-	-	-	43.0	-	-	-	76.0	-
ArrayD	-	-	-	90.0	-	-	-	82.0	-	-	-	73.0
	Bandwidth				**Throughput**				**Latency**			
Relative Fitness	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
ArrayA	1	92.0	98.0	25.0	1	62.0	54.0	54.0	1	38.0	45.0	66.0
ArrayB	32.0	1	34.0	73.0	39.0	1	34.0	26.0	20.0	1	89.0	102.0
ArrayC	85.0	43.0	1	60.0	74.0	95.0	1	58.0	86.0	98.0	1	98.0
ArrayD	55.0	56.0	46.0	1	36.0	76.0	23.0	1	40.0	37.0	34.0	1

Table 7.13: Tree sizes (leaf nodes) and their averages.

Consider an extreme case where one array is always half as fast as another, regardless of the workload. Then, only one relative fitness value (0.5) is necessary to model the performance of the slow array relative to the faster one. In fact, the diagonal in Table 7.13 for the relative fitness models shows the size of a tree when an array is compared with itself (i.e., each model has one leaf node with a ratio of 1.0).

In practice, relative fitness models must learn different relative fitness values, depending on the workload. In the worst case, each workload type will have a different relative fitness value, and a relative fitness model may learn a rule for each. However, it is often the case that different workloads share the same relative fitness value, thereby resulting in much smaller trees. When averaged over all trees in this evaluation, the relative fitness models, when compared to the absolute models, are 26% smaller for bandwidth, 25% smaller for throughput, and 20% smaller for latency.

7.9.4 Summary

Relative fitness models were constructed to measure the benefits of predicting performance ratios rather than performance values. In support of Hypothesis 5, Experiment 5 can be summarized as follows:

Result 5 *The best predictors of the relative fitness models are the performance observations, followed by the read characteristics, and then the write characteristics. When compared to the relative performance models, the relative fitness models can reduce the median relative error of the bandwidth predictions up to 33% (*`FitnessCache` *is reduced from 47% to 14%), throughput up to 23% (*`FitnessCache` *is reduced from 36% to 13%), and latency up to 6% (*`FitnessDirect` *is reduced from 20% to 14%). When compared to the absolute models, bandwidth error is decreased up to 53% (*`FitnessCache` *is reduced from 67% to 14%), throughput up to 23% (*`FitnessCache` *is reduced from 36% to 13%), and latency up to 20% (*`FitnessDirect` *is reduced from 34% to 14%). Overall all applications, the relative fitness models reduce the average median relative bandwidth prediction error from 23% to 11%, throughput from 17% to 10%, and latency from 21% to 14%.*

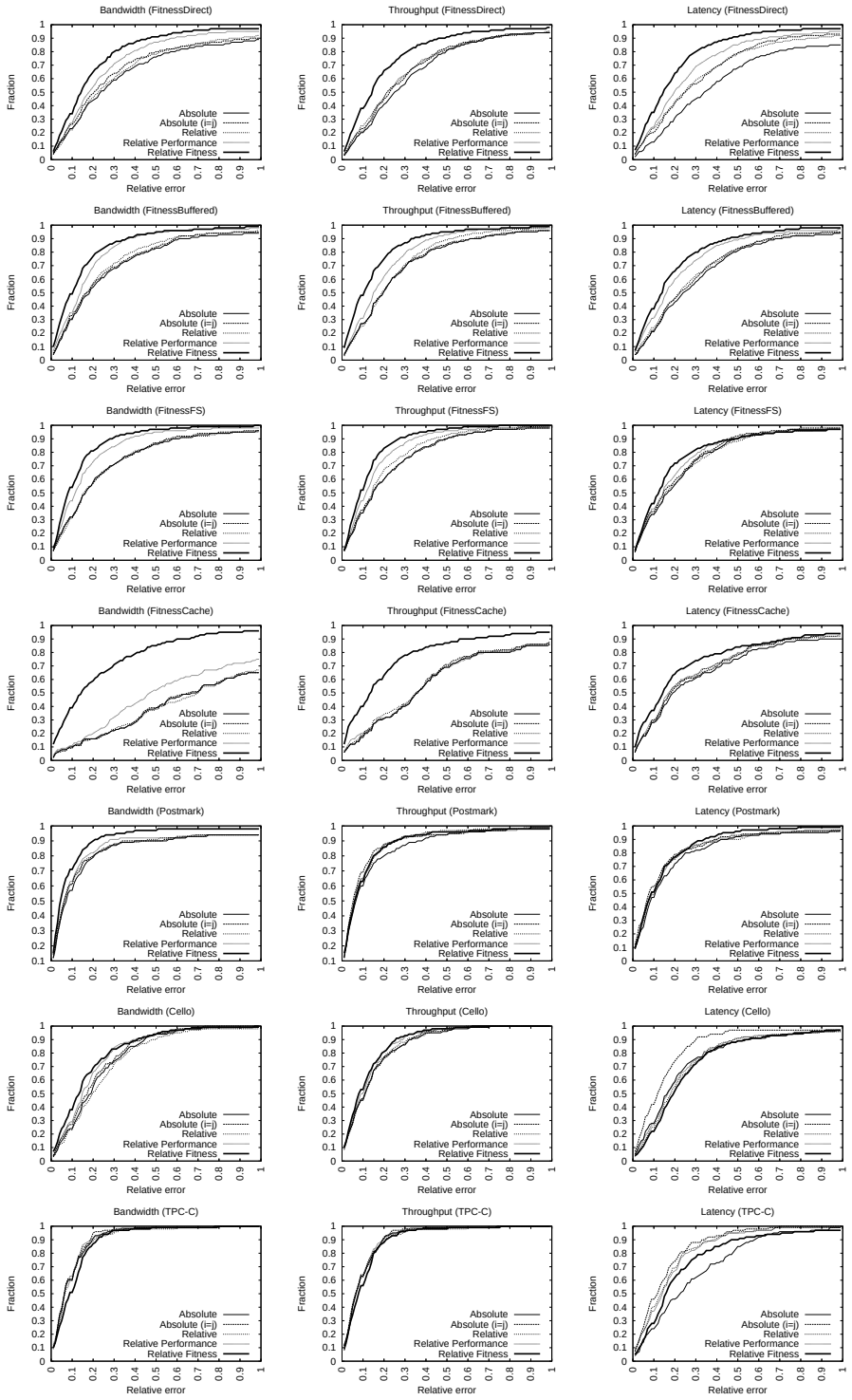

Figure 7.33: Relative error CDFs.

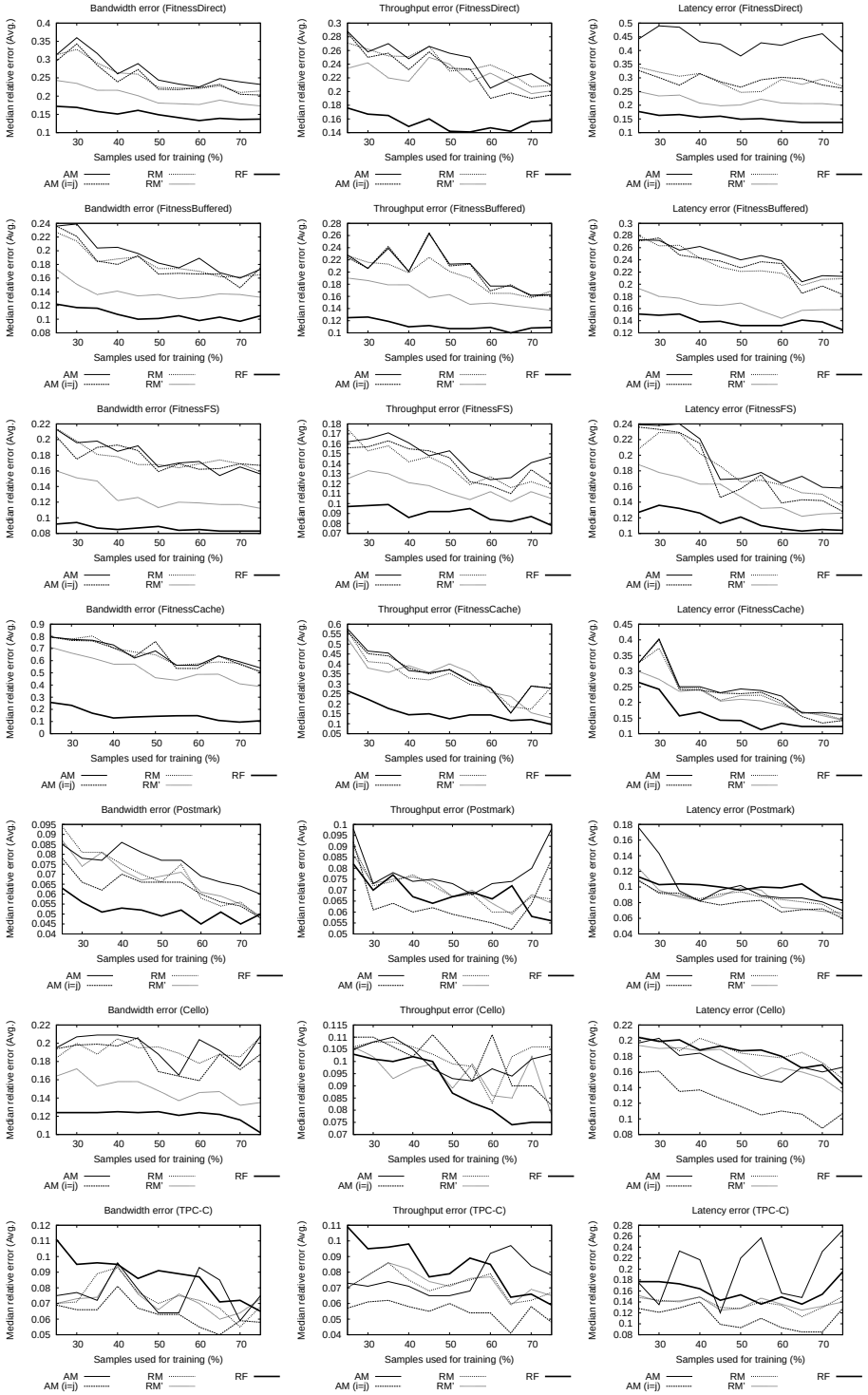

Figure 7.34: Error vs. training set size. The training set size varies from 25% of the collected samples to 75%; the remaining samples are used for testing. For each training set size, an error metric (median relative error) is calculated for each pairing of arrays. These metrics are then averaged across all pairings to produce the values shown.

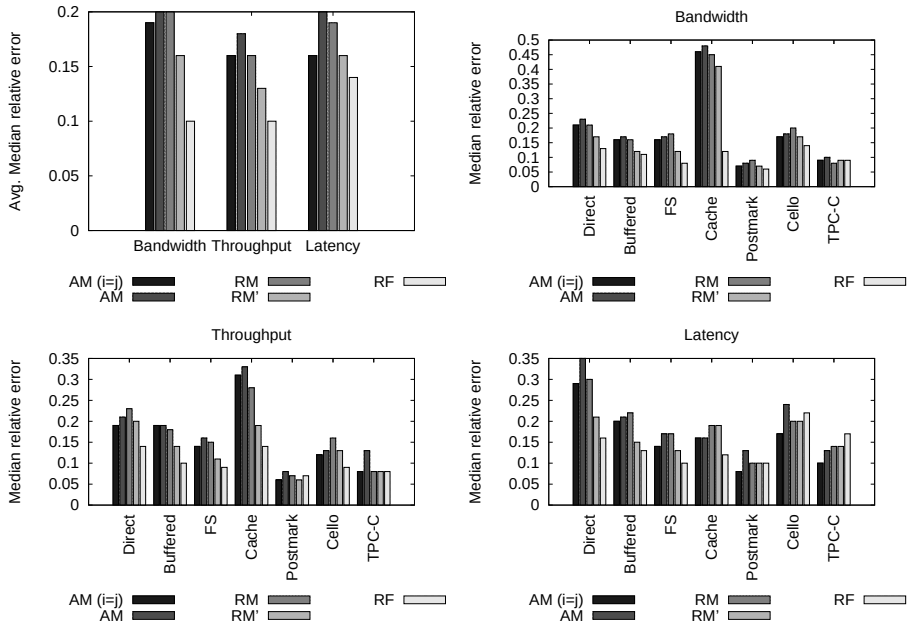

Figure 7.35: Median relative error: Experiment 5 (mixed models). The graphs show the median relative error of each application, comparing the idealized absolute model, titled "AM (i=j)," with the absolute model (AM), the relative model (RM), the relative performance model (RM'), and the relative fitness model (RF). Also shown are the average median relative errors of the applications for each performance metric.

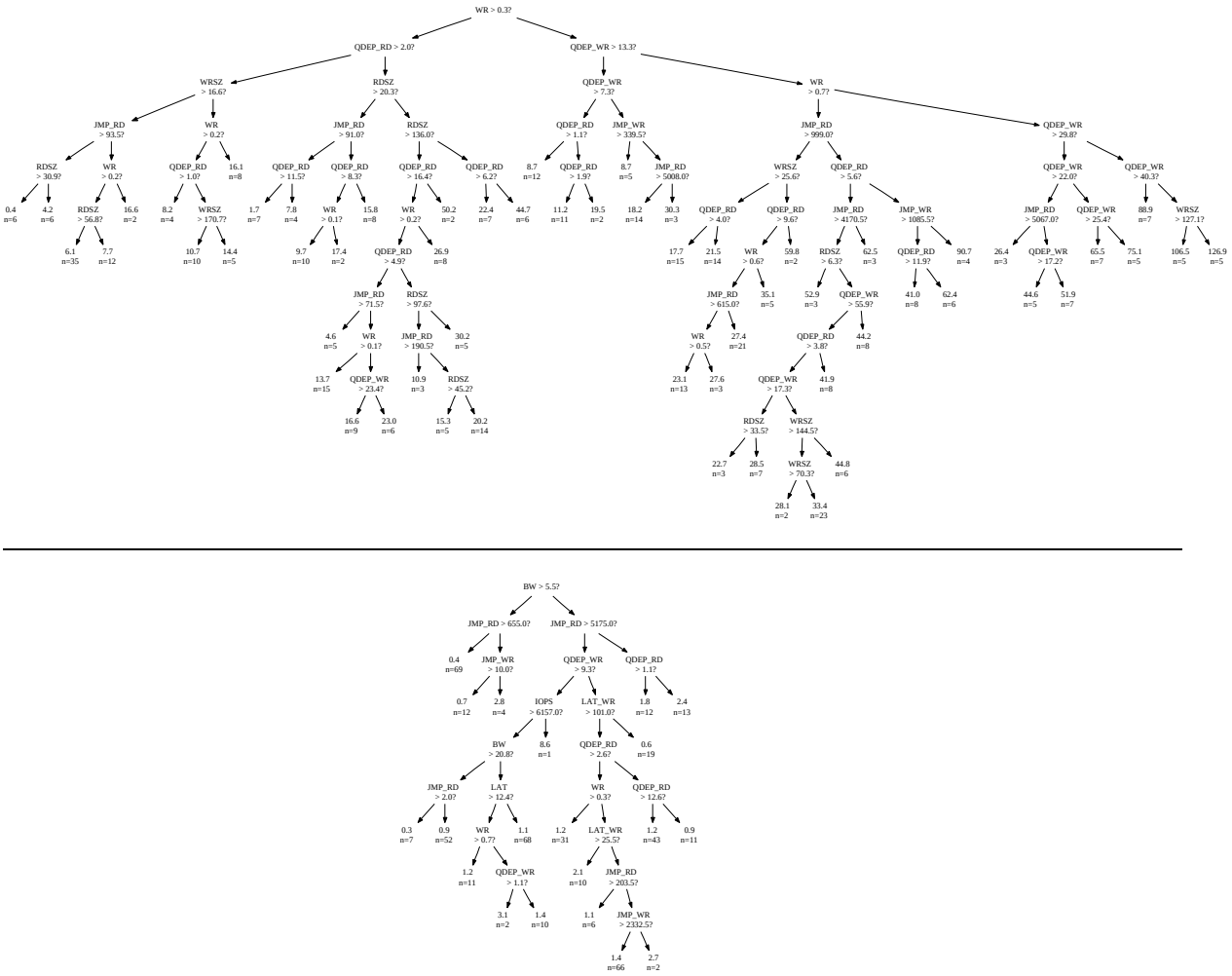

Figure 7.36: The absolute latency model of Array A for `WorkloadMix` (top) versus the relative fitness model for Array $B \rightarrow A$ (bottom). These are the actual models with no additional pruning for readability. The absolute model has 59 leaf nodes (or predictions rules) versus 20 rules for the relative fitness model. In other words, the relative fitness model learns the latency of Array A, relative to Array B, with approximately one third as many rules as that required by the absolute model.

Bibliography

[1] Network Appliance. PostMark: A New File System Benchmark. http://www.netapp.com.

[2] Leo Breiman, Jerome Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Chapman and Hall, New York, NY, 1984.

[3] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *Proceedings of the International Conference on Management of Data (ACM SIGMOD 1994)*, Minneapolis, MN, May 24–27 1994. ACM Press.

[4] Intel Corporation. Open Storage Toolkit. http://www.sourceforge.net/projects/intel-iscsi.

[5] Intel Corporation. Storage System SSR212MA. http://www.intel.com.

[6] Intel Corporation. Storage System SSR316MJ2. http://www.intel.com.

[7] Transaction Processing Performance Council. TPC Benchmark C. http://www.tpc.org/tpcc.

[8] EqualLogic. PS100E Storage Array. http://www.equallogic.com.

[9] Eleftherios Koutsofios and Stephen C. North. Drawing graphs with *dot*. Technical report, AT&T Bell Laboratories, October 1993. Dot User's Manual.

[10] Hewlett-Packard Laboratories. Cello 1999 traces. http://www.hpl.hp.com/research/ssp.

[11] LeftHand Networks. iSCSI Disk. http://www.lefthandnetworks.com.

[12] Open-E. iSCSI Enterprise. http://www.open-e.com.

[13] Julian Satran. Internet SCSI (iSCSI). http://www.ietf.org/rfc/rfc3720.txt.

[14] Salford Systems. CART. http://www.salfordsystems.com.

Chapter 8

Summary and future work

Storage management is an expensive component of data center administration. Automated storage management could help reduce costs by offloading many time-consuming and error-prone tasks. One such task is storage system design. Numerous optimization-based methods have been explored, many of which require fast and accurate performance predictions. Conventionally, absolute performance models have been used to make these predictions. However, researchers have wrestled with a necessary ingredient of such models (workload characterization) for more than three decades.

The fundamental challenge with workload characterization is compressing an I/O workload into a concise set of workload characteristics without losing performance-affecting information. A second challenge imposed by absolute models is the requirement that the characteristics be absolute. However, as shown by Experiment 2, a variety of workload characteristics can change due to the closed-loop interactions between an operating system and a storage system. Also, an additional challenge for black-box models is achieving sufficient coverage over the space of workload characteristics so that training workloads can accurately predict the performance of new workloads. Relative fitness models are designed to help address each of these challenges.

First, the closed-loop relationship between an operating system and storage system can be learned by modeling storage systems relative to one another, thereby reducing the prediction error due to changing workload characteristics. This dissertation's results show that the write characteristics are more likely to change than the read characteristics (Experiment 1), that these changes increase the prediction error of an absolute model (Experiment 2), and that a relative model can significantly reduce this prediction error (Experiment 3). For example, the median relative error of the `TPC-C` latency predictions for the array pairing $B \rightarrow C$ increased from 11% to 46% due to changing characteristics, and a relative model reduced this error to 17%.

Second, the dependence on workload characteristics can be reduced by also using performance observations. For the relative performance models (Experiment 4), it was shown that a workload's observed bandwidth on one storage system is the best predictor of its bandwidth on another storage system, as well as for throughput and latency, and that training a model to use observed performance can reduce prediction error. For example, the relative performance model reduced the median relative error of the `FitnessBuffered` latency predictions (over all array pairings) from 23% to 16%.

Third, one can reduce the amount of required training data for a statistical, black-box performance

model by predicting performance ratios (relative fitness values) rather than performance itself. It was shown that relative fitness models (Experiment 5) require less training data for an equivalent, or better, level of prediction accuracy, when compared to absolute, relative, and relative performance models. For example, when compared to a relative performance model, the median relative bandwidth prediction error for `FitnessCache` was reduced from 67% to 14%.

Over all predictions in this study, when compared to the absolute models, the relative fitness models reduced the median relative bandwidth prediction error from 23% to 11%, throughput from 17% to 10%, and latency from 21% to 14%. This result supports the thesis statement that one can improve the prediction accuracy of statistical, black-box models by modeling storage systems relative to one another.

8.1 Future work

Additional research is required to quantify the benefits of relative fitness modeling with respect to reducing storage management costs. This would include integrating relative fitness models into an optimization framework for automated storage system design. And, there are many other interesting areas of future work, including the use of workload interference models, resource utilization, domain knowledge, cost functions, collaborative filtering techniques, machine learning models (other than CART), and active learning. Each of these is described below.

First, storage consolidation may require that multiple workloads share the same storage system, so the effect of resource contention among multiple I/O streams is important to model. Appendix B introduces the potential for predicting whether workloads will interfere with one another, but much more work is required in this area.

Resource utilization is an area not explicitly explored in this work, as this would have required modifying the storage systems (e.g., to retrieve cache statistics). However, a workload's observed resource utilization could provide additional information to the relative fitness models.

The models presented in this dissertation are black-box and require no knowledge of a storage system's internals. However, domain knowledge may produce more accurate grey-box models. Examples of domain knowledge include a description of a storage system's hardware/software platform (e.g., cache size, cache eviction algorithms, number of disk drives), its configuration (e.g., the RAID level), and its expected performance behavior for various workload types.

One could explore the risk/reward trade-off when making predictions. Although relative fitness models have the ability to make a prediction for any workload, the confidence of any such prediction can vary considerably (e.g., 95% versus 75%). Such confidence levels could be consulted when deciding whether to move application data from one storage system to another. In some cases, the cost of a misprediction may be too high.

By using multiple relative fitness models, techniques based on collaborative filtering could further improve prediction accuracy. Such a scenario is possible when a workload has been observed by more than one storage system. Rather than use a single model, one could explore ways of combining the predictions of multiple models.

This dissertation presented relative fitness modeling in the context of CART, but any number of machine learning algorithms could be used. For example, an early follow-on to this work (A. Zheng, M.

Mesnier, and C. Faloutsos) indicates that relative fitness models based on nearest neighbor algorithms produce lower relative error than equivalent models based on CART.

Finally, a large cost of black-box modeling is the time required to obtain representative training data. In this work, a synthetic workload generator was used to randomly sample (uniformly) the large space of possible workloads, but doing so required many hours of workload generation. Such sampling does not base future samples on the performance of past samples. However, more *active* approaches to sample collection could potentially reduce the amount of required training time, by more intelligently selecting training samples. Moreover, as described in Section 2.4, the performance of real, truly representative, workloads – as they are introduced into the data center – could be used as additional training data if the models fail to accurately predict their performance. Using the same modeling techniques presented in this work, the characteristics and performance of such workloads could be used to construct new models. One could also explore machine learning techniques that would weigh real workloads more heavily than synthetic workloads.

Appendix A

Open Storage Toolkit

The software used in this dissertation is based on the 2002 iSCSI/OSD open source release from Intel. The original code was extended with support for workload characterization and synthetic I/O generation (Fitness). All code has been re-released as Intel's Open Storage Toolkit.

This appendix contains the Fitness command-line arguments and scripts used to generate the workload samples, based on version 2.0.18 of the Open Storage Toolkit.

A.1 `Fitness` usage

```
Usage: fitness [options]
 --file <file>            Data file (e.g., /mnt/data)
 --no_create              Assume file already exists (dflt 0)
 --allow_sparse           Do not zero-fill file before doing I/O (dflt 0)
 --device <device>        Target device (e.g., /dev/sd0, /dev/raw/raw0)
 --mount <dir>            Mounts an ext2 on <device> and then creates <file>
 --direct                 O_DIRECT flag for <file> or <device>
 --config <file>          ips.conf file (dflt /etc/ips.conf)
 --tid <num>              Target id (specify -1 if not iSCSI)
 --lun <num>              iSCSI lun (dflt 0)
 --samples <samples>      Number of samples (default 1)
 --skip <skip>            Samples to skip over (default 0)
 --iters <iters>          Iterations per sample (default 1)
 --seed <seed>            Seed for RNG (default 1234)
 --flush <cmd>            Command to run after each iteration
 --flush_args <args>      Args to pass to flush command
 --breather <secs>        Time to rest after each iteration (dflt 0)
 --cap <capacity>         Working set size in MB (default 128)
 --disp <capacity>        Displacement from lba 0 in MB (default 0)
 --wr <num>               Percent writes (default rnd)
 --str <1|2>              Shared or separate rd/wr streams (default rnd)
 --qdep <num>             Queue depth (default rnd)
 --wrsz <num>             Write size (KB) (default rnd)
 --rdsz <num>             Read size (KB) (default rnd)
 --wr_stride <num>        Write stride (KB) (default 0)
 --rd_stride <num>        Read stride (KB) (default 0)
 --wrnd <num>             Percent random writes (default rnd)
 --rrnd <num>             Percent random reads (default rnd)
```

```
 --wrsk <dist>            Seek (KB) distribution per write I/O
 --rdsk <dist>            Seek (KB) distribution per read I/O
 --think <num>            Think time (default rnd)
 --warm <secs>            Warm time in secs (default 10)
 --warm_io <num>          Warm for this many IOs
 --warm_mb <num>          Warm for this many MB of data
 --test <secs>            Test time (default 10)
 --term <capacity>        Stop test after <capacity> MB bytes transferred
 --outfile <outfile>      Output file for results (default stdout)
 --mock                   Don't actually do any I/O
 --extern <cmd>           Run command for each iteration, instead of sampling
 --extern_args <args>     Pass these args to the external command
 --extern_suppress        Suppress stdout and stderr from external command
 --extern_shutdown        Run this command to shutdown external command
 --extern_mark <string>   Start warming after <string> encountered in stdout
 --replay <trace>         Replay udisk trace (see udisk -t option)
 --timing_accurate        Attempt a timing accurate trace replay
 --filter                 Apply sample filter hardcoded in fitness.c
 --i_counters             Retrieve initiator counters after each iteration
 --i_counters_loop <secs> Retrieve initiator counters every <secs> secs
 --version                Show version
 --help                   Show usage
```

A.2 Command for `--flush` option

This script is used to flush the storage array caches after each workload sample is collected. It reads 1 GB of data sequentially (twice) at an offset of 16 GB. The script is stored in the file `~/usr/bin/doflush`:

```
#!/bin/sh

fitness --device /dev/sd0 --tid 0 --cap 1024 --disp 16384 --term 1024 \
--wr 0 --rdsz 64 --qdep 16 --rrnd 0 --iters 2 --direct
```

A.3 Options for `FitnessDirect`

```
fitness --tid 0 --device /dev/sd0 --i_counters --cap 16384 --warm 30 --test 30 \
--samples 200 --iters 3 --flush ~/usr/bin/doflush --direct
```

A.4 Options for `FitnessBuffered`

```
fitness --tid 0 --device /dev/sd0 --i_counters --cap 16384 --warm 30 --test 30 \
--samples 200 --iters 3 --flush ~/usr/bin/doflush
```

A.5 Options for `FitnessFS`

```
fitness --tid 0 --device /dev/sd0 --i_counters --cap 16384 --warm 30 --test 30 \
--samples 200 --iters 3 --flush ~/usr/bin/doflush --file /mnt/data --mount /mnt
```

A.6 Options for FitnessCache

```
#!/bin/sh

for cap in 16 16384; do for qdep in 1 2 4 8 16; do for rdsz in 1 2 4 8 16 32 64; do \
fitness --tid 0 --device /dev/sd0 --direct --i_counters --warm 30 --test 30 \
--iters 3 --samples 1 --flush ~/usr/bin/doflush --wr 0 --rrnd 0 \
--qdep $qdep --rdsz $rdsz --cap $cap; \
done; done; done
```

A.7 Options for Postmark (creation phase)

```
!#/bin/sh

for i in `seq 0 49`; do \
fitness --tid 0 --device /dev/sd0 --i_counters --cap 16384 \
--warm 60 --test 30 --iters 3 --samples 1 --skip $i \
--flush ~/usr/bin/doflush \
--mount /mnt --extern postmark \
--extern_args ~/rf/etc/postmark-create.$i.cfg \
--extern_suppress; \
done
```

A.8 Options for Postmark (transactions phase)

```
!#/bin/sh

for i in `seq 0 49`; do \
fitness --tid 0 --device /dev/sd0 --i_counters --cap 16384 \
--warm 60 --test 30 --iters 3 --samples 1 --skip $i \
--flush ~/usr/bin/doflush \
--mount /mnt --extern postmark \
--extern_args ~/rf/etc/postmark-trans.$i.cfg \
--extern_suppress \
--extern_mark transactions; \
done;
```

A.9 Options for Cello

```
#!/bin/sh

for i in `seq 0 79`; do \
fitness --tid 0 --device /dev/sd0 --i_counters --cap 16384 \
--warm 999 --warm_mb 128 --test 30 --samples --iters 3 \
--flush ~/usr/bin/doflush --replay tracefile.$i.txt
done
```

A.10 Options for TPC-C

```
#!/bin/sh
```

```
for i in `seq 0 49`; do \
fitness --tid 0 --device /dev/sd0 --i_counters --cap 16384 \
--warm 30 --test 30 --samples 1 --iters 3 \
--mount /mnt \
--flush ~/usr/bin/doflush \
--extern ~/usr/bin/run-tpcc --extern_suppress \
--extern_args $i --extern_mark START; \
done
```

A.10.1 Contents of ~/usr/bin/run-tpcc

```
#!/usr/bin/perl

$SIG{INT} = sub {
    $pid = getpgrp(0);
    system("iscsi-kill-tree $pid");
};
$skip = $ARGV[0];
$dir = "/mnt";

# copy in a configuration file with randomly generated values for
# prob_neworder, prob_payment, prob_order_status, prob_delivery and prob_stock_level
system("cp ~/rf/etc/exampleconfig.$skip ~/usr/src/tpcc-kit/exampleconfig");

if (fork()!=0) {
    while (1) {
     sleep(1);
    }
} else {
    system("mkdir $dir/tpcc-log");
    system("cd $dir; /h/ss/dist/benchmark-tpcc/tpcc-kit/main -s0");
    system("cd $dir; /h/ss/dist/benchmark-tpcc/tpcc-kit/main -s1 -sm_logging n");
    system("sync");
    printf("START\n");
    system("cd $dir; /h/ss/dist/benchmark-tpcc/tpcc-kit/main -r0");
}
```

A.10.2 Contents of ~/usr/bin/tpch-run

```
if (fork) {
    system("rm -f /tmp/tpch.COPYDONE");
    system("echo y | sudo mkfs /dev/sd0 16777216 > /dev/null");
    system("sudo mount /dev/sd0 /mnt");
    system("sudo chmod a+w /mnt");
    system("cp -Rv ~/rf/cache/tpch/data /mnt > /dev/null");
    system("touch /tmp/tpch.COPYDONE");
    system("postmaster -D /mnt/data > /dev/null");
    system("sudo umount /mnt");
    system("rm -f /tmp/tpch.COPYDONE");
} else {
    do {
        sleep(1);
    } until (-f "/tmp/tpch.COPYDONE");
```

```
    sleep(10);
    system("echo parent: RUNNING PSQL");
    system("psql tpch < ~/usr/src/tpch_postgres/var/queries/$ARGV[0]-ready.sql > /dev/null");
}
```

A.10.3 Contents of ~/usr/bin/tpch-shutdown

```
#!/bin/sh

killall psql
sleep 1
killall postmaster
```

Appendix B

Concurrent workloads

This experiment models the performance impact of two workloads running concurrently on the same disk array. The hypothesis is that a regression tree can be used to predict whether two workloads, running together, will yield more/less bandwidth than the first workload running alone. Throughput or latency could have also been used.

Of course, when multiple workloads are sharing a disk array, one really wants to predict the quality of service that each workload will receive (e.g., the proportion of the bandwidth), rather than the aggregate performance of the array. As a first step toward this goal, this experiment establishes that one can use workload characteristics to predict the relative change in aggregate performance when another workload is added. If aggregate performance is predicted to decrease, then one has established the minimum (predicted) performance degradation that the first workload will experience. Longer term, it may be possible to use such models recursively. For example, if two workloads are running and characterized as a single workload, one might be able to predict the performance impact of adding a third workload, and so on.

The performance metric used is referred to, in this experiment, as an *impact factor*, or the amount by which aggregate bandwidth changes when the second workload is added to the first. Values less than 1.0 indicate an adverse impact, and values greater than 1.0 indicate a performance improvement. For example, if a single workload has a performance of 100 MB/sec, and adding a second workload reduces the aggregate bandwidth to 75 MB/sec, the impact factor is 0.75.

B.1 Setup

The same hardware and software platforms, as described in Section 7.4.1, are used for this experiment. Each disk array is modeled independently (i.e., no relative modeling).

Concurrent runs of Fitness are used to generate pairs of synthetic workloads. The application-level workload parameters of Fitness are used to characterize each workload. As described in Section 7.4.2, these include the write percent (0 to 100), write size (KB), read size (KB), write randomness (0 to 100), read randomness (0 to 100), and queue depth. Because application-level parameters are used as workload characteristics, there is no change in workload characteristics across disk arrays. Further, the arrays are accessed in raw mode (no page cache or file system).

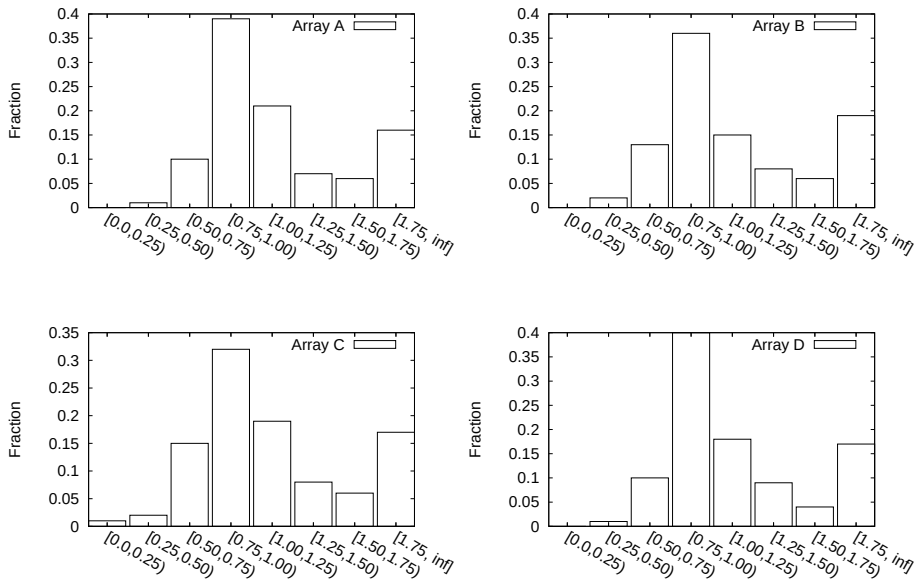

Figure B.1: Distribution of performance impact on each disk array.

1000 pairs of synthetic workload are generated for each disk array. The first workload of each pair is run first, and the performance is observed. The second workload is then added, and the aggregate bandwidth of the two workloads is observed. From these observations, the impact factor is calculated as previously described.

B.2 Distributions of performance impact

Figure B.1 plots the distribution of the performance impact for each disk array, over all 1000 workload pairs. Two observations can be made. First, in approximately 50% of the cases, on each disk array, adding a second workload to the first increases aggregate bandwidth. In the remaining 50%, adding a second workload results in a less efficient I/O, thereby leading to a reduction in bandwidth. In other words, the performance impact depends on the type of workloads composing the pair. Second, as indicated by the differing distributions, a pair of workloads can react differently across arrays. Some arrays may show an adverse performance impact, others may not. Together, these two observations suggest that modeling the performance impact on a disk array will require that a) one use workload characteristics (and possibly performance) and b) one create separate models for each disk array.

B.3 Performance models

Using 50% of the workload pairs, CART models are trained to predict the performance impact. The model inputs are 1) the workload characteristics and performance of the first workload on a given disk array and 2) the workload characteristics of the second workload.

Table B.1 shows the median relative error for each disk array. These values represent the relative

	Naive Predictor	CART Predictor
Array A	48%	14%
Array B	62%	13%
Array C	66%	18%
Array D	49%	16%

Table B.1: Error when predicting the change in aggregate bandwidth when a second workload is added to a disk array.

error when predicting the impact factor. Also shown is a naive model that predicts the average impact factor over all training workloads. As can be seen in the table, a CART model can significantly improve the prediction error over a naive model, thereby suggesting that predicting the effects of resource sharing is possible with CART.

B.4 Discussion

This experiment established the potential for modeling the performance impact of concurrent workloads. A more general hypothesis, but one not tested in this experiment, is that this process can be applied recursively. That is, if n workloads are running on a disk array, their aggregate workload characteristics and performance can be observed, and a CART model can be used to predict whether adding an additional workload will negatively impact the overall bandwidth. In such a scenario, assuming workloads receive a proportional/fair share of resources, one can continue adding workloads to a disk array as long as the aggregate performance continues to increase. This methodology is similar to continuing to admit processes into a multi-user system, so long as the system is not "thrashing."