Generating and Analyzing Synthetic Workloads using Iterative Distillation

A Thesis
Presented to
The Academic Faculty

by

Zachary A. Kurmas

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
May 2004

Generating and Analyzing Synthetic Workloads using Iterative Distillation

Approved by:

Dr. Umakishore Ramachandran, Adviser

Dr. Kimberly Keeton
(Hewlett-Packard Laboratories)

Dr. Kenneth Mackenzie
(Reservoir Labs, Inc.)

Dr. Yannis Smaragdakis

Dr. Panagiotis Manolios

Date Approved: 14 May 2004

To Candi, for her love, support,

and many years of patience.

ACKNOWLEDGEMENTS

Those who praise "thinking outside the box" have never had to deal with a graduate student who chose a thesis topic outside every one of his department's "boxes." Many people have gone above and beyond the call of duty to provide me the resources and advice needed to finish this dissertation. First, I want to thank Kimberly Keeton, Ralph Becker-Szendy, John Wilkes, Allistar Vietch, and everybody else in the Storage Systems Department at HP Labs for granting me the privilege to join them for a summer, and for continuing to support my research after I had returned to Georgia Tech. Second, I want to thank Ann Chervenak, Ken Mackenzie, Kishore Ramachandran, and all the other faculty at Georgia Tech for providing the moral and financial support that allowed me to pursue my storage systems research. Third, I thank Chad Huneycutt, Josh Fryman, and all the other students at Georgia Tech who spent time editing my papers and critiquing my talks. Nobody was under any obligation to support my research, yet they agreed to work together and see that I was able to complete my dissertation.

I would also like to acknowledge the incredible support staff in the College of Computing at Georgia Tech. Barbara Binder, Jennifer Chisholm, Cathy Dunahoo, Barbara Durham, Deborah Mitchell, Linda Williams, and the rest of the administrative assistants do an excellent job helping students navigate Georgia Tech's typical university bureaucracy. Neil Bright, Karen Carpenter, and the rest of the Computing and Networking Services staff do a wonderful job maintaining the College of Computing's computing infrastructure and shielding students from the overhead of system administration. These two groups of people have saved me from countless hours of frustration and wasted time.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The exponential growth in computing capability and use has produced a high demand for large, high-performance storage systems. Unfortunately, advances in storage system research have been limited by (1) a lack of evaluation workloads, and (2) a limited understanding of the interactions between workloads and storage systems. We have developed a tool, the *Distiller* that helps address both limitations.

Our thesis is as follows: Given a storage system and a workload for that system, one can automatically identify a set of workload characteristics that describes a set of synthetic workloads with the same performance as the workload they model. These representative synthetic workloads increase the number of available workloads with which storage systems can be evaluated. More importantly, the characteristics also identify those workload properties that affect disk array performance, thereby highlighting the interactions between workloads and storage systems.

This dissertation presents the design and evaluation of the Distiller. Specifically, our contributions are as follows. (1) We demonstrate that the Distiller finds synthetic workloads with at most 10% error for six out of the eight workloads we tested. (2) We also find that all of the potential error metrics we use to compare workload performance have limitations. Additionally, although the internal threshold that determines which attributes the Distiller chooses has a small effect on the accuracy of the final synthetic workloads, it has a large effect on the Distiller's running time. Similarly, (3) we find that we can reduce the precision with which we measure attributes and only moderately reduce the resulting synthetic workload's accuracy. Finally, (4) we show how to use the information contained in the chosen attributes to predict the

performance effects of modifying the storage system's prefetch length and stripe unit size.

CHAPTER 1

INTRODUCTION

We have developed a tool, the *Distiller* that automatically searches for those workload characteristics (formally called *attribute-values*) that two block-level workloads must share in order to have the same behavior on a given storage system. This dissertation presents the design of the Distiller, evaluates its ability to find useful attribute-values, explores its configuration options, and discusses several ways that it can help overcome some of the current difficulties in storage systems research.

1.1 Motivation

The exponential growth in computing capability and use has produced a high demand for large, high-performance storage systems. The storage systems research community has done a commendable job developing new hardware and software to meet this demand. However, two related limitations have hindered their progress: (1) a lack of evaluation workloads and (2) a limited understanding of the interactions between workloads and storage systems. The Distiller will help researchers address both limitations.

The behavior of large enterprise storage systems depends heavily upon the choice of workload. Consequently, researchers must evaluate potential design decisions and self-configuration algorithms using workloads that represent how the storage system will be used in a production environment.

One approach for driving storage system studies is to use block-level traces from an actual storage system in production use. However, as described in [21], using traces of real storage system activity has a number of limitations:

1. Traces are difficult to obtain, often for non-technical reasons. System administrators are reluctant to permit tracing on production systems; and when they do collect traces, it is often time-consuming to anonymize them to protect users' privacy.

2. Although any single trace file may not be huge, a set of trace files describing the activity of a system over a longer period of time (weeks or months) may occupy considerable space (tens of gigabytes), making them difficult to store on-line and share over the Internet.

3. It is difficult to isolate and modify specific workload characteristics of a trace (e.g., arrival rate, total accessed storage capacity, distribution of request locations). Consequentially, traces do not support explorations of slightly larger, busier, burstier, or other hypothetical future workloads.

The alternative approach is to use synthetic workloads. A synthetic workload is randomly generated in a way that preserves the key characteristics of some realistic target workload. We will use the terms *attribute* and *attribute-value* to discuss a workload's characteristics. An attribute is a metric for measuring a workload (e.g., mean request size, read percentage, or distribution of location value). An attribute-value is an attribute paired with the quantification of that attribute for a specific workload (e.g., a mean request size of 8KB, or a read percentage of 68%). Chapter 3 contains formal definitions of these terms.

Synthetic workloads do not suffer from the limitations of workload traces:

1. We can describe a set of synthetic workloads using only values of the target workload's key attributes. These attribute-values do not contain user-specific information; therefore, companies should have few concerns about making such attribute-values publicly available.

2. We have found attribute-values that describe a reasonably accurate synthetic workload and, when optimized, are an order of magnitude smaller than the corresponding compressed workload trace.

3. By adjusting the summarized attribute-values, we may be able to modify the resulting synthetic workload so it approximates future workloads.

The challenge is that we do not know precisely which attribute-values a synthetic workload must share with the target workload on which it is based. If we do not choose the attributes carefully, the synthetic workload will not be *representative of* (i.e., will not behave like) the target workload. At a high level, we know that two workloads must have (among other things) the same degrees of spatial locality, temporal locality, and burstiness; however, we do not know how precisely to quantify and reproduce these properties.

The literature provides a number of attributes for quantifying locality, burstiness, and other important workload characteristics [21, 24, 25, 32, 33, 55, 56]; however, each of these attributes is targeted for, and tested on, a particular type of workload (e.g., database, file system). As a result, synthesizing a representative workload requires an impractically time consuming and tedious process of searching the related work for a set of attributes that sufficiently describes the specific workload under study.

1.2 Overview of the Distiller

Our solution was to develop a tool, the Distiller, to automate this tedious process. The Distiller, when given a workload trace and a library of possible workload attributes, automatically determines which attribute-values the target workload must share with the synthetic workload in order to be representative. These attribute-values serve as a *compact representation* of the synthetic workload.

The Distiller takes as input a *target* workload trace and a library of attributes. It then either finds a subset of attributes from the library whose values specify a

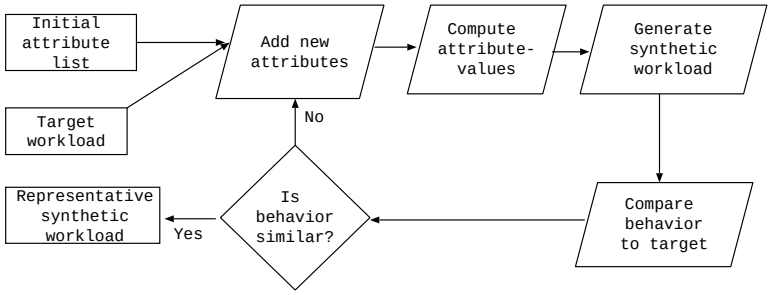

Figure 1: Distiller's iterative loop

synthetic workload representative of the target, or determines that such a specification is not possible using the attributes in the library. Synthetic workloads must maintain the specified set of attribute-values.

At a high level, the Distiller iteratively builds a list of "key" attributes — attributes that noticeably influence the target workload's behavior. During each iteration, the Distiller identifies one additional key attribute (from the library of candidate attributes), adds it to the list, then tests the representativeness of the resulting synthetic workload by replaying it on the storage system under test and comparing the resulting response time distribution to that of the target workload. This loop (shown in Figure 1) continues until either (1) the difference between the behavior of the synthetic and target workloads falls below some user-specified threshold, or (2) the Distiller determines that it cannot specify a more representative synthetic workload by adding a small number of attributes from the library.

One approach is to simply add and evaluate attributes in an arbitrary order until the synthetic workload meets the stopping conditions. However, this approach will not work because groups of attributes often have offsetting effects. As a result, a synthetic workload based on a single additional attribute may be less representative than a synthetic workload based on two or more additional attributes. Therefore, we need a method of evaluating individual attributes apart from their immediate effects when added to the list of key attributes.

Evaluating all possible sets of attributes would address this problem; however, this

approach is too time consuming. Even if each evaluation required only milliseconds, a library of a few hundred attributes would make an exhaustive search intractable.

Instead, we developed a method of using only two evaluations to determine whether an entire group of related attributes contains any "key" attributes. We partition attributes into fifteen groups (one group for each non-empty subset of {operation type, location, interarrival time, request size}), then estimate the benefit of adding the most useful attribute in each group (i.e., the attribute that leads to the most representative synthetic workload) to the current attribute list. If the best possible attribute in a given group has little or no effect on the behavior of the workload, then we assume that no attribute in the group is a key attribute and do not explore the attribute group further. Instead, we focus on only those groups which, according to our test, contain at least one key attribute. In addition, our technique of dividing attributes into groups allows us to evaluate an individual attribute with respect to its group instead of its effects when added to the key attribute list. Our method allows us to use divide-and-conquer and "best-first"-like techniques in our search for key attributes. Chapter 5 contains a detailed definition of the attribute groups and the evaluations used to test attribute groups and individual attributes.

1.3 Contributions

The primary contribution of this thesis is the Distiller and the resulting ability to efficiently generate representative synthetic workloads. We designed and implemented a novel technique of automating the tedious trial-and-error aspect of developing representative synthetic workloads. This automation makes synthetic workload generation practical. The Distiller evaluates attributes in an intelligent order and avoids exhaustively searching all possible combinations of attributes. Furthermore, if the Distiller is unable to produce a representative synthetic workload, it lists the attribute groups for which the library contains no useful attributes. By identifying the group to which a

missing attribute belongs, the Distiller helps direct the development of new attributes, when necessary.

Our study of the Distiller provides four additional, related contributions:

1. **An evaluation of the Distiller:** We evaluate (1) the quality of the synthetic workloads the Distiller specifies, (2) the size of the synthetic workload's compact representations, and (3) the number of evaluations the Distiller performs during its execution. Given the current library of attributes, the Distiller can specify a synthetic workload with at most 10% error for all but three workloads evaluated. In the worst case, the resulting synthetic workload has a 25% error. During its initial run, the Distiller specifies synthetic workloads that have compact representations that are typically 25% to 60% the size of the compressed original workload trace itself. It can then post-process the chosen attributes and further reduce the compact representation to between 1% and 10% of the compressed original workload trace. During each run, the Distiller generates and evaluates between 15 and 125 workloads.

2. **A study of the Distiller's configuration decisions:** Before running the Distiller, the user must choose (1) which error metric to use when comparing workload behavior and (2) a threshold for exploring attribute groups and choosing attributes. We investigate the effects of these configuration decisions on the accuracy of the synthetic workloads the Distiller specifies, the size of their compact representations, and the number of evaluations the Distiller performs during its execution.

   We found that the error metric has a very large effect on the attributes the Distiller chooses and the accuracy of the resulting synthetic workload. Overall, a metric called "hybrid", led to the most accurate synthetic workloads. The hybrid error metric combines the differences between the mean response times

and the differences between the overall shape of the response time distribution graphs. Considering the overall distribution of response times allows the Distiller to estimate how well the chosen attributes capture different behaviors, such as cache or tail (i.e., long-latency I/O) behavior. However, many people will not consider a synthetic workload to be similar to the workload it models unless both workloads have similar mean response times. The accuracy threshold has a smaller effect on the accuracy of the resulting synthetic workload, but has a larger effect on the size of the attributes the Distiller chooses and the number of workloads it generates and evaluates.

3. **A study of the tradeoff between size and accuracy:** We study the tradeoff between the compactness of a synthetic workload's representation and its accuracy. The Distiller specifies synthetic workloads that have compact representations that are typically 25% to 60% the size of the original workload trace itself. Fortunately, in many cases, reducing the precision of the chosen attributes produces a synthetic workload with similar accuracy, but with a much more compact representation. For example, reducing the size of a synthetic Email server workload's compact representation by 70% had no affect on its accuracy. Reducing the size of a synthetic OLTP database workload's compact representation by 5% actually doubled its accuracy.

4. **An analysis of the usefulness of the synthetic workloads specified by the Distiller:** We show that the synthetic workloads specified by the Distiller are accurate enough to allow evaluators to choose between several competing design decisions. Specifically, we show that the attributes chosen by the Distiller contain enough information to predict the effects of changing the storage system's prefetch length and stripe unit size.

1.4 Organization

The remainder of this thesis is organized as follows: Chapter 2 discusses the background and related work. Chapter 3 defines our workload models, the attributes we use to characterize workloads, and the metrics we use to compare workload behavior. Chapter 4 discusses our hardware, software, and workload traces. Chapter 5 provides the details of the Distiller's design and implementation. Chapters 6 through 8 present our evaluations; and Chapter 9 concludes. In addition, Appendix A discusses our techniques for generating synthetic workloads; and Appendix B provides the algorithms we use to implement the Distiller's various demerit figures.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter discusses the historical context of our work and how our approach complements existing workload characterization and synthetic workload generation techniques. Both of these areas have their roots in performance evaluation. Sections 2.1 through 2.3 discuss performance evaluation, workload characterization, and synthetic workload generation, respectively. Section 2.5 discusses our contributions to these areas; and Section 2.4 discusses related approaches.

2.1 Performance evaluation

Scientists have studied techniques for evaluating computer systems since the early 1960's when "The need for standardized performance criteria [was] emphasized by the wide range of commercially available computers, the varied configurations each can assume, the greater variety of tasks and methods of implementing them, and the ever-present limitations upon time and money for the selection, programming and operation of the system [31]." In the early days of computing, scientists were primary concerned with developing usable hardware. However, the increasing variety of technology available necessitated an understanding of the advantages and disadvantages of the different technologies. This understanding is important not only for the consumer, who must choose between competing products, but also for the scientists, who must choose between competing design decisions or research directions.

The advent of multi-programming and multi-processing computer systems complicated the process of performance evaluation. The primary performance metric of a batch processing system was throughput: how quickly the machine could complete

a given job [34]. The introduction of multi-programming systems encouraged the development of interactive applications, such as airline reservation systems, for which the response time of individual requests was also important.

The increase in complexity of computers and applications during the 1950s and 1960s further complicated the field of performance evaluation. The diversity of computer applications combined with the diversity of hardware units such as channels, control units, storage, tape, and tape drives created a need for many different performance measurement techniques because "no single number can serve adequately as the measure of the performance capability of a piece of equipment, of a program module, or even a completed system [12]." The growing complexity of computer systems also increased the importance of choosing the test workload carefully. Instead of choosing a test workload based primarily on its CPU load, it became necessary to consider the effects of the workload on all hardware components. Furthermore the increase in the number and diversity of hardware components created a need for the evaluation of the individual components themselves.

Ferrari's performance evaluation books from the 1970s and 1980s [17, 20] discuss the important factors in evaluating a computer system including:

- different modeling techniques, specifically simulation models, analytical models, and observing physical devices;
- different measurement techniques (hardware vs. software and continuous vs. sampling) and their effects on the workload and the measurements obtained;
- different techniques of generating loads, including traces, stochastic models, and deterministic models; and
- a classification system for workloads used to drive evaluation studies.

According to Ferrari, the workload used to drive an evaluation is only a model of the "real" workload that the system under test will serve in real life (unless the

system is evaluated while in production use). He asserts that even complete workload traces are only models of the real workload.

Ferrari's most important contribution to our research is his discussion of methods with which evaluators judge the quality of the test workloads they use to drive system evaluations. Like Calingaert [12], he notes that there is no single metric with which a workload model can be judged; however, he emphasizes that, in general, evaluators should judge test workloads using *performance-oriented* criteria. By "performance-oriented", Ferrari means that we should base the measure of similarity between the test workload and the real workload on how much the results of the intended performance evaluation differ from the results of the same evaluation had it been run using the real workload. In contrast, "cosmetic" comparison criteria compare only the characteristics of the two workloads. Such comparisons judge how well the test workload maintains the desired similarities, but do not judge how well the chosen characteristics capture the performance-affecting aspects of the real workload.

2.2 Workload characterization

Workload characterization is a fundamental aspect of performance evaluation [19]. The performance of any system (storage system, mainframe computer, database, etc.) depends on its workload. Thus, we must present the results of any evaluation within the context of a workload. In order to understand that context, we must describe, or *characterize* the workload.

Initially, researchers used workload characterization primarily to specify the class of workload they used to evaluate a system. These characterizations were simple: They either divided jobs into classes by function (e.g., compiler, scientific problem, business problem) or described jobs based on their utilization of resources such as processor, I/O channels and devices, core memory, and unit record devices (number of cards read, lines printed, etc.) [49].

The increasing complexity of computers necessitated that evaluators describe evaluation workloads more precisely with respect to their effects on each system component. For example, the introduction of virtual memory made a program's page fault behavior a critical aspect of the performance of the system under test [11, 15, 18]. Furthermore, the growing complexity also increased the precision with which evaluators described the mix of jobs and their interactions [13, 14].

As computers became more complex, scientists also began to study individual subsystems, including the I/O subsystem. An I/O workload represents thousands or millions of requests from different applications, where each application may produce a different pattern of requests. As explained in Section 3.2, the patterns in the requests can have a very large effect on the I/O subsystem's performance. Consequently, instead of characterizing workloads simply to classify them, researchers began to characterize workloads to help understand them and their effects on design decisions under consideration.

For example, Ousterhout, et al., characterized a UNIX 4.2 BSD file system workload with the goal of collecting "information that would be useful in designing a shared file system for a network of personal workstations [41]." They measured the distribution of file sizes, file lifetimes, throughput, and cache hit ratios, then demonstrated that a reasonably sized cache could eliminate enough of the network traffic to make a networked file system feasible. Ousterhout's work is the first of many I/O trace studies [6, 7, 10, 22, 26, 39, 44, 45] designed to provide an intuitive understanding of a workload.

2.3 Synthetic workload generation

Synthetic workload generation is, in some sense, an extension of workload characterization. The purpose of a workload characterization is to describe a workload using properties that are of interest to the user; the purpose of synthetic workload

generation is to then provide a workload with the desired properties.

Researchers often combine workload characterization and synthetic workload generation. When generating performance evaluation workloads for computer systems, the standard approach, as outlined in [19], is to characterize individual jobs, then use clustering or other statistical techniques to choose a representative sample of jobs. The synthetic workload is the list of chosen sample jobs [13, 14, 49].

In addition to general computer system workloads, researchers use the sampling approach to generate several other types of workloads, including memory and instruction traces (for workloads to drive processor evaluations) [5, 37, 59], and, to a limited extent, file system workloads [51]. The sampling technique, however, is not well-suited for synthesizing block-level I/O workloads. Clustering- or sampling-based synthesis techniques often assume that each component (job) consumes a fixed amount of resources (CPU time, I/O bandwidth, memory, etc.) each time it is issued. This assumption is not true in a storage context because the effects of locality, caching, and prefetching can make the resources consumed for an I/O request highly variable. Thus, to use clustering- or sampling-based techniques to synthesize I/O workloads, one must define components to be something other than a single I/O request. Section 2.3.2 provides examples of such clustering-based generation techniques for I/O workloads.

In response to the difficulties of using existing techniques, many researchers have developed other synthetic generation techniques. These techniques fall into three categories: (1) modeling the source, (2) reproducing patterns, and (3) reproducing behavior.

2.3.1 Model the source

Generation techniques that model the source attempt to behave like the user and/or application that generated the workload being modeled. For example, SynRGen

generates file system workloads by using "micromodels" to simulate the I/O activity of different applications [16]. For example, the micromodel of a compiler reads several files in their entirety (e.g., the .c and .h files), then creates another file (the .o file) and writes to it. SynRGen also models different users by stochastically switching among different application models. For example, the model of a user in an edit/debug cycle iterates between an Emacs micromodel and a compiler micromodel.

Similarly, many benchmarks are designed to exhibit typical behavior for a given application. The TPC-C benchmark issues queries and other commands typical for an on-line transaction processing (OLTP) database [43, 38]. The TPC-H benchmark issues queries typical for a decision support database [42, 60]. The Standard Performance Evaluation Corporation (SPEC) has developed benchmarks that simulate the load for several different types of servers, including World Wide Web servers, Email servers, and file servers [50]. Although TPC and SPEC did not design these benchmarks specifically to provide representative I/O workloads, executing each benchmark produces an I/O load that can be traced and used as a workload model.

The challenge in synthesizing a workload by modeling user or application behavior is that modeling behavior is fundamentally the same problem as directly modeling the I/O workload (i.e., sequence of I/O requests), except instead of explicitly specifying the I/O workload characteristics that should be present in the synthetic workload, the user specifies the characteristics of the user behavior to be reproduced. In many cases, developing an accurate user- or application-level model is just as difficult as generating the synthetic block-level I/O workload.

2.3.2 Reproduce the pattern

The second group of workload generation techniques explicitly attempt to produce a workload with a given set of patterns or characteristics. Most synthetic I/O generation techniques, including ours, fall into this category [9, 21, 24, 25, 26, 32, 33, 35, 55, 56].

Although these generation techniques seek only to reproduce a specified pattern, the design and expected behavior of the disk array (as described in Section 3.2) often motivates the choice of patterns reproduced. The remainder of this section discusses these synthetic generation techniques.

Distribution sampling: The simplest generation technique is to choose the values for a request parameter independently at random from some distribution. In the past, scientists assumed that interarrival times followed a Poisson distribution, location values followed a uniform distribution, and request sizes followed a normal distribution. They constructed an implicit distribution based on only a mean value and standard deviation, then drew values from this distribution regardless of its similarity to the modeled workload's actual distribution. Ganger demonstrated that the assumptions on which scientists based these implicit distributions were incorrect and that using the incorrect assumptions led to unrepresentative synthetic workloads [21].

Using the distribution in the modeled workload produces a more accurate synthetic workload, but rarely produces a representative synthetic workload because it does not measure and reproduce the correlations (relationships) that usually exist among request parameters [20]. For example, most workloads exhibit locality of reference, which is a correlation between location values. Similarly, many workloads are bursty, which is a correlation between interarrival times. Bodnarchuk and Bunt, however, successfully used this technique to generate a file system workload over NFS [9].

β-model: The β-model seeks to generate a synthetic workload that maintains the same level of "burstiness" as the modeled workload [56]. Wang, et al., based it on the "80/20" law for databases, which states that 80% of queries involve only 20% of the data. Given m I/Os, and a parameter β (where $.5 \leq \beta \leq 1$), the β-model allocates βm I/Os to one randomly chosen half of the trace (first half or second half). It then allocates the remaining $(1-\beta)m$ I/O to the other half and recursively repeats

this process on each half of the trace.

The parameter β is based on the slope of the entropy plot. Specifically, for each of n aggregation levels, the β-model divides the trace into 2^n intervals by time and computes the entropy of those intervals. Entropy at level n is defined to be

$$E(n) = -\sum_{i=1}^{2^n} p_i \log p_i$$

where p_i is the fraction of requests in each interval i. The parameter β is related to the slope of the line generated by plotting $E(n)$ against n. Specifically,

$$\text{slope} = -\beta \log_2 \beta - (1-\beta) \log_2 (1-\beta)$$

Hong and Madhyastha modified this technique slightly by using the multifractal spectrum to estimate β [32].

PQRS: Wang, et al., also developed an attribute that "captures all the characteristics of real spatio-temporal traffic" [55]. In other words, this technique strives to not only reproduce the burstiness of the access pattern and the arrival pattern, but also to reproduce the correlations between them. The PQRS algorithm measures four parameters (p, q, r, and s) that are based on the joint entropy of the location and arrival time values. The corresponding generation technique then uses these values to recursively construct a joint distribution for location and arrival time. (The recursive construction is the two-dimensional equivalent of the β-model's construction.)

Cluster-based trace synthesis: Hong, Madhyastha, and Zhang developed a generation technique that chooses several intervals of a real trace to represent the entire trace. The algorithm then generates a complete synthetic trace by concatenating copies of those intervals [33].

Their method generates a synthetic workload as follows: First, it divides the target workload into intervals of some constant length (Hong and Madhyastha experimented with intervals ranging from .1 seconds to 10 seconds) and characterizes the intervals

by number of requests, aggregation ratio (the ratio of non-empty bins to empty bins), and entropy (using a similar technique to the β-model). The algorithm then clusters the intervals using an agglomerative hierarchal clustering technique and chooses one representative interval for each cluster. Finally, the algorithm produces a synthetic workload by concatenating copies of the cluster representatives.

An agglomerative hierarchal clustering technique begins by placing each item (e.g., interval) in its own cluster. It then iteratively agglomerates (i.e., merges) the most similar clusters until the desired number of clusters remain.

Gomez access pattern: This generation technique [24, 25] assumes that the trace of the I/O workload being modeled identifies the process that issued each request and that these processes generate I/Os in an ON/OFF pattern (i.e., that they produce several I/O in a short amount of time, then are silent for some amount of time).

Within an ON period, Gomez's algorithm chooses a location in one of three ways:

- sequential to the previous I/O,
- equal to the starting location for the current ON period, or
- spatially local to (i.e., within 500 sectors of) to the starting location for the current ON period.

Gomez and Santonja base their method of choosing a particular location on measurements of the target workload. The algorithm also uses similar techniques to choose the location of the first request in an ON period; however, in this case the algorithm not only considers the process's previous location, but also the location of the previous ON period's first request.

Gomez ON/OFF generator: This generator produces a self-similar arrival pattern by simulating and combining the arrival pattern of different processes [24]. It first analyzes the sources (processes) in the target workload. Gomez and Santonja call those sources that exhibit an ON/OFF pattern (alternating periods of much activity

and no activity) "permanent sources." They call those sources that are active only for a short time with no long periods of inactivity "vanishing sources." To model a permanent source, the algorithm draws the length of the ON times and OFF times from heavy-tailed distributions. The model for the vanishing sources "was proposed by Cox based on an $M/G/\infty$ queuing model where customers arrive according to a Poisson process and have service times drawn from a heavy-tailed distribution with infinite variance." In all cases, the generator defines specific distributions based on corresponding mean values in the modeled workload (mean ON time, mean OFF time).

2.3.3 Reproduce behavior

The third group of generation techniques do not attempt to generate a workload with specific characteristics. Instead, they attempt to generate each request such that it causes the system to exhibit a specific behavior (e.g., cache hit, or a specific response time). We are unaware of any generation techniques that actually attempt to reproduce behavior; however, we provide here a hypothetical description of such a technique.

The input to this generation technique would be a distribution (or possibly a list) of response times for individual I/O requests. Then, for each request, the generation technique would consider the current state of the disk array and choose the request's parameter values to produce an I/O with the desired response time. For example, if the next I/O needed a very short response time, the generator would construct an I/O to be a cache hit. The generator would also adjust the interarrival time to avoid queuing delays. Similarly, to generate an I/O with a very long response time, the generator would construct the I/O to be a cache miss and/or choose a very short interarrival time to cause a queuing delay.

Developing such a technique will be very difficult because it requires a detailed

model of the disk array that can very accurately predict the response time. In addition, it may not be possible to generate arbitrary lists of response times. For example, generating extremely long response times may require generating considerable queue lengths. It may not be possible to build up the queue while simultaneously generating many very low latency I/Os. Generating a queue requires planning several I/Os in advance. Such planning may not be computationally feasible.

Although a generator that accurately reproduces individual response times may not be practical, we can easily write a generator containing a cache simulator that specifically generates I/O locations that hit or miss in the cache as desired. If the generator wishes to generate a cache hit, it can have the simulator generate a location in the cache. This generator does not completely reproduce behavior, however, because it is unaware of other influences such as queue length. Varma and Jacobson used this approach in a study of de-staging algorithms for disk arrays [53].

2.4 Attribute selection

The Distiller is fundamentally a modified best-first search technique. We will see in Chapter 5 how, during each step of its iterative technique, the Distiller chooses the attribute that it estimates will provide the best partial solution. Feature selection and principal component analysis are two other techniques that researchers use to select attributes.

Feature selection techniques choose a set of features (attributes) that can differentiate workloads by performance. The Artificial Intelligence field has defined several standard feature selection techniques [8]. The challenge is that these techniques require a large training set (i.e., many workloads) to explore. Currently, our set of available workloads is too small. Also, researchers use standard feature selection techniques to choose attributes that will distinguish the workloads in the example set. If the example set does not cover the entire space of workloads, the feature

selection technique may disregard a performance-related attribute because no other workloads happen to have sufficiently different values for that attribute.

Principal component analysis (PCA) is another technique for defining a small set of attributes. Researchers have used principal component analysis to identify the attribute-values that describe a set of batch computational workloads [14]. Given a large set of workloads and a set of attribute-values that characterize those workloads, PCA computes new variables, called principal components, that best describe the set of workloads. These principal components are uncorrelated linear combinations of the original attribute-values. Choosing attributes in this manner presents several challenges. First, using PCA to identify performance-related attributes requires a set of workloads with similar performance; we do not currently have access to such a set. Second, the resulting principal components are linear combinations of attributes rather than a subset of the initial attribute list and, hence, may not have any intuitive meaning.

2.5 Our work

The Distiller's major contribution is that it automatically incorporates a performance-oriented criterion into its technique for choosing the characteristics (i.e., attribute-values) on which it bases synthetic workloads. In contrast, researchers based the synthetic workload generation techniques in the literature on workload attributes they chose *a priori* using their domain expertise. These choices may be valid for some workloads and storage systems but not for others. Instead of presenting another synthetic workload generation technique, the Distiller leverages these existing techniques to automatically choose the ones that are most appropriate for the target workload and storage system under test. The set of current block-level analysis and corresponding generation techniques serve as the Distiller's "library" of candidate attributes. We are not aware of any technique for automatically selecting attributes

used to characterize and synthesize I/O workloads.

The Distiller also provides a similar contribution to the field of workload characterization. As with synthetic workload generation, researchers tend to choose attributes to study workloads *a priori*. These attributes are often useful for providing an intuitive "picture" of the workload; however, they may or may not be related to the workload behavior under study. Using the Distiller to select attributes can assure that the chosen attributes are relevant.

Finally, we expect the Distiller to contribute to the evaluation of storage systems by providing additional test workloads. It is currently difficult to obtain accurate test workloads. System administrators hesitate to make workload traces publicly available because they are concerned that the traces contain sensitive data that could be mined and used against the company. The attribute-values with which we specify synthetic workloads contain high-level information that should not be specific enough to compromise a business's profitability. Thus, we expect that companies will be more willing to provide attribute-values that evaluators can use to construct synthetic workloads than they will be to provide complete workload traces. The challenge is determining which attributes to use — precisely the problem we designed the Distiller to solve.

CHAPTER 3

WORKLOAD MODELS, ATTRIBUTES, AND PERFORMANCE METRICS

This chapter provides definitions and background to help the reader understand our workload models, analysis techniques, and performance comparison metrics. Section 3.1 describes our workload model and provides several terms we use when defining specific workload attributes. Section 3.2 helps the reader understand the issues involved in evaluating disk arrays and the motivation behind several of the attributes in the Distiller's library by discussing the relationship among the disk array, a workload, and the performance of the workload on the disk array. Section 3.3 defines the attributes that we have implemented and placed in the Distiller's library. Finally, Section 3.4 discusses our metrics for comparing workload behavior.

3.1 Workload

This section defines our workload model and provides definitions for terms we use to define attributes.

3.1.1 Our workload model

A block-level workload for a storage system is a sequence of individual I/O requests. Each request has four parameters:

- **Operation type:** A request's *operation type* is either "read" or "write."
- **Location:** The *location* parameter is a logical value that identifies the location of the data in the storage system. In general, an I/O's location includes both

Table 1: Example workload

	I/O workload				
I/O number	**Operation type**	**Location**	**Request size**	**Arrival time**	**Interarrival time**
1	Read	1024	8192	0	NA
2	Read	9216	8192	.001	.001
3	Read	17408	1024	.003	.002
4	Write	25600	8192	.004	.001
5	Write	18432	2048	.009	.005
6	Read	20480	4096	2.66	2.57
7	Write	19456	1024	2.69	.003
8	Write	51200	8192	7.87	5.18

a device number (which identifies either a physical disk logical partition of the storage system) and a logical address on that device. We present this pair implicitly using one value; however, it can also be presented explicitly using two values. Because location is a logical value, locations x and $x + 1$ are not necessarily physically adjacent; they may lie on separate tracks or separate disks.

- **Request size:** The *request size* is the number of bytes requested.

- **Arrival time:** The time at which a request is issued is its *arrival time*. Some workloads present the *interarrival time* instead of the arrival time. The interarrival time is the time elapsed since the previous request's arrival time. The choice of whether to present arrival time or interarrival time is a matter of convenience when using an open workload model because each set of values can be calculated directly from the other.

The source of the requests is not relevant to this workload model. We can obtain a list of requests from a trace of a production storage system or generate the list randomly using a synthetic workload generator. Table 1 presents a sample workload and its interarrival times.

Table 2: Example jump distances and modified jump distances

I/O number	Operation type	Location	Request size	Arrival time	Jump distance	Modified jump dist.
1	Read	1024	8192	NA	NA	NA
2	Read	9216	8192	.001	0	8192
3	Read	17408	1024	.002	0	8192
4	Write	25600	8192	.001	7168	8192
5	Write	18432	2048	.005	-15630	-7168
6	Read	20480	4096	2.57	0	2048
7	Write	19456	1024	.003	-5120	-1024
8	Write	51200	8192	5.18	32768	31744

3.1.2 Open vs. closed model

Section 3.1.1 presents an *open* workload model. An open model specifies the issue time of each request relative to either the beginning of the trace (arrival time) or to the issue time of the previous request (interarrival time). For this thesis, we use only open workload models.

The other common workload model is the *closed* model, which includes the CPU time between I/O requests issued by the same thread. In a closed model, an I/O's issue time may depend on the completion time of a previous I/O; therefore, this model specifies an I/O's issue time relative to the completion time of the last synchronous I/O issued by the current thread.

In general, closed models are more accurate than open models. Consider a set of I/O requests that a single thread issues synchronously with no processing time between. Each I/O's issue time depends upon the previous I/O's response time. The open model does not reflect this dependency.

3.1.3 Describing workload attributes

In this section, we define several common terms that we use to describe workload attributes.

- **Jump distance:** The *jump distance* between two I/Os is the distance in the location address space from the end of one I/O to the beginning of the next.

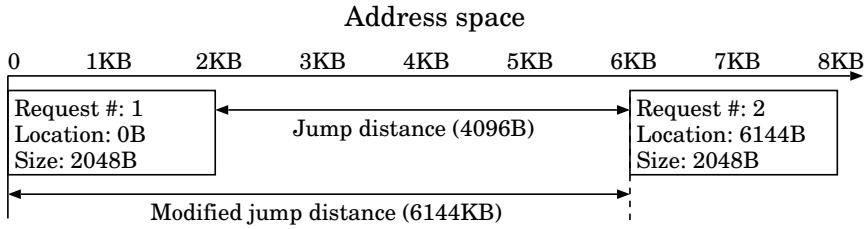

Figure 2: Jump distance

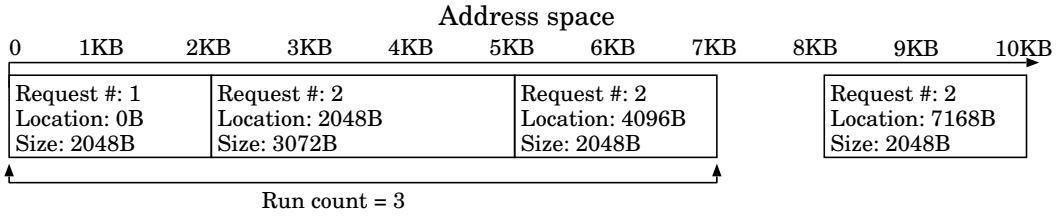

Figure 3: Run count

For example, in Table 2, the jump distance between I/Os 3 and 4 is $25600 - (17408 + 1024) = 7168$. The jump distance between I/Os 4 and 5 is $18432 - (25600 + 8192) = -15630$. Figure 2 provides a visual example of jump distance.

- **Modified jump distance:** The *modified jump distance* between two I/Os is the distance in the location address space from the beginning of one I/O to the beginning of the next. For example, in Table 2, the modified jump distance between I/Os 3 and 4 is $25600 - 17408 = 8192$. Figure 2 provides a visual example of modified jump distance. (The Distiller's search technique occasionally studies each sequence of I/O request parameter values independent of the other request parameters. Using modified jump distances allows us to study a workload's location values apart from its request sizes.)

- **Run count:** A *run* is a sequence of I/Os for which the first byte of each I/O immediately follows the last byte of the previous I/O (i.e., a jump distance of 0). For example, I/Os 1 - 3 in Table 2 form a run with a run count of 3. I/Os 5 - 6 form a run with a run count of 2. Figure 3 provides a visual example of run count.

- **Modified run count:** A *modified run* is a sequence of I/Os for which the modified jump distances are all equal. For example, I/Os 1 - 4 in Table 2 form a modified run with a modified run count of 4 because their location values are all 8192 bytes apart. (The Distiller's search technique requires that we occasionally study each sequence of I/O request parameter values independent of the other request parameters. Using modified run counts allows us to study a workload's location values apart from its request sizes.)

- **Burstiness:** Researchers consider a workload's arrival pattern to be *bursty* if there are some periods of time containing many I/O requests and other periods of time containing very few, if any, requests. There is currently no universal definition of burstiness. Instead, there are several attributes that capture different aspects of burstiness. Examples include β from the β-model (defined in Section 3.3) and the coefficient of variation of interarrival time.

- **Footprint:** The set of logical addresses accessed at least once by a workload is its *footprint*. The footprint of the workload in Table 1 is

  $[1024, 24576) \quad [25600, 33792) \quad [51200, 59392)$

  This footprint has a size of 39936 bytes. Notice that not every location accessed is the beginning location of an I/O. For example, the first I/O accesses location 2048.

3.2 Disk arrays and performance

This section discusses the relationship among the disk array, a workload, and the performance of the workload on the disk array. Understanding the components and organization of the disk array will help the reader understand which workload patterns affect behavior and help motivate the attributes in the Distiller's library. Understanding how disk arrays and workloads interact also provides insight into the challenges

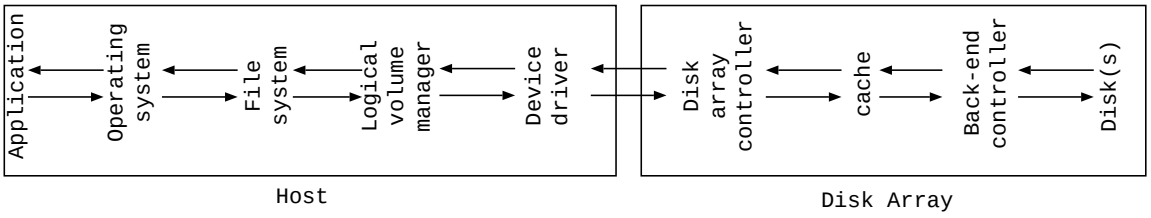

Figure 4: The path of a typical I/O from the application through the disk array

of generating representative synthetic workloads and accurately predicting disk array performance.

In this section, we discuss the HP FC-60, which has a design similar to most modern, high-end disk arrays [29]. Our FC-60 contains thirty 18GB Seagate ST118202LC disks spread evenly across six disk enclosures. It has two disk array controllers in the same controller enclosure with a 40 MB/sec Ultra SCSI connection between the controller enclosure and each of the six disk enclosures. Each controller can access all of the SCSI buses, and has 256 MB of battery-backed cache (NVRAM) [36]. (Table 7 in Section 4.1 provides the configuration details for each disk array we use.)

First, we will describe the path a typical read request takes from the application through the FC-60. Figure 4 illustrates this path. Applications use a system call to make I/O requests. The operating system services that system call by issuing it to the file system. If the file system's cache does not satisfy the request, the file system issues it to the logical volume manager (LVM) which identifies the logical volume that contains the requested data. The LVM then makes a request using the disk array's device driver. The device driver communicates with the FC-60's disk array controller over a Fibre Channel link. The disk array controller first checks to see if the disk array's cache can service the request. If so, it obtains the data from the cache and sends it back up the path to the application. Otherwise, the cache requests the data from the back-end controller which determines exactly which disk or disks contain the data, requests the data from the disks, assembles it if necessary, and sends it back up the path to the application.

Each I/O can potentially be queued at any step along this path. The I/Os that must be queued somewhere tend to have the longest response times. A long queue can develop in the host if the operating system's buffers are full. A long queue can also develop in the disk array controller if the array receives more requests in a short amount of time than the disks can serve, or if a request requires data from a disk that happens to be busy.

Most writes in a write-back caching environment have very low latency. Each disk array controller has a 256MB write-back cache. Consequently, the disk array always places the data of a write request into the cache and writes it to disk at a later time (a process called *de-staging*). The FC-60's write-back cache uses non-volatile RAM; therefore, the data is safe, and the request is complete, as soon as the disk array's cache receives it. (The exception is when the write buffer is full. In this case, the disk array immediately de-stages enough data from the cache to make room for the new request. The new write request is not complete until the "foreground" de-staging is complete.) Requests for data stored in the cache have a very low latency because access to memory is at least two orders of magnitude faster than access to disk. Notice, however, that because the array must eventually de-stage written data, the writes may contribute to the queuing delays of future requests.

If the cache does not contain the requested data, the disk array controller must determine which disk or disks contain the data and issue a request for it. Identifying the physical location of data is not a trivial task. A disk array's individual disks are organized into *logical units*. Each logical unit (LU) implements a RAID redundancy group and often appears to the user as a single storage device. The disk array *stripes* data over the disks that comprise a logical unit to allow several disks to serve the request in parallel. In addition, the disk array may *mirror* (i.e., copy) data onto two or more disks to provide fault tolerance.

The logical unit to which a system administrator assigns a set of data (e.g., a file

system, or a database table) has a large effect on performance. For example, two database tables that are often accessed together should be assigned to different LUs (to avoid contention), whereas two tables that are almost never accessed together should share a single LU (so that the disks are better utilized). The problem of configuring disk arrays for optimal utilization and throughput is open and currently under study [2, 3, 4].

The disk array controllers may make several optimizations when servicing requests. First, they can carefully order requests to the disk drives to reduce bus contention and minimize seek and queuing time. Similarly, they can carefully schedule the destaging of dirty cache data so that the data is written when the disk heads pass over the data's physical location on their way to read other data. Also, the disk array can *prefetch* data — request the next several kilobytes of data in anticipation of the user requesting that data soon. These optimizations make assumptions about patterns in the workload (for example, that data tends to be requested sequentially). Workloads that exhibit the expected patterns tend to have lower request times than those that do not.

The individual disks themselves can perform many of the same optimizations as the disk array controllers. Many modern disk drives have caches, reorder logic, write-behind, and prefetch capabilities. The interactions of a workload and a single disk have been extensively studied and are fairly well understood [46, 48]. Unfortunately, placing many disks together in a single device complicates the interactions [40].

In order to specify an accurate synthetic workload, the attributes in the Distiller's library must capture the patterns in the workload that stress each of the components along the I/O path. For example, the resulting synthetic workload must have the same number of cache hits as the target workload, must cause queues to build in the same way, and must cause the disk heads to spend similar amounts of time seeking and reading data. These factors dominate the behavior of modern disk arrays; however,

with the advent of new technology (such as MEMS-based storage devices [52]), other factors may play a prominent role in disk array behavior.

3.3 Attributes and attribute-values

An *attribute* is a metric used to measure a workload characteristic (e.g., mean request size, read percentage, or distribution of location value.) An *attribute-value* is an attribute paired with the measurement itself for a specific workload (e.g., a mean request size of 8KB, or a read percentage of 68%). If one views an attribute as a function, f, then an attribute-value is the pair $(f, f(x))$ for some workload trace x. An *analyzer* is an algorithm that computes an attribute's value for a given workload.

Attributes describe a measurement of only the workload itself; they do not measure the response of the underlying storage system when subjected to the workload. For example, "mean response time" is not a valid attribute. Furthermore, attributes must be fully defined. "Locality" and "burstiness" are not valid attributes because there are many different ways to quantify locality and burstiness. In contrast, β from the β-model is a valid attribute.

The remainder of this section defines the attributes that we have implemented and placed in the Distiller's library, discusses how we organize those attributes within the library, and explains how we define the size of an attribute-value.

3.3.1 Distributions

The Distiller's library contains two types of empirical distributions: Joint distributions and conditional distributions.

Empirical distribution: We use the term *empirical distribution* to refer to the actual, explicit distribution of values for some I/O request parameter. Figure 5 shows the example workload's distribution of request size.

We use *histograms* to present empirical distributions. A histogram partitions a parameter's range into subsets (called *bins*) and lists the number of I/Os that

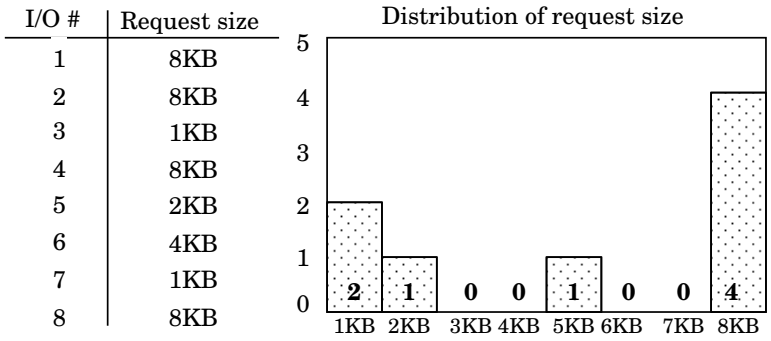

Figure 5: Distribution of request size for example workload

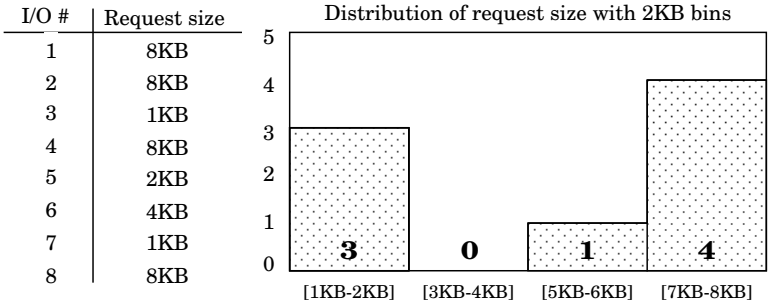

Figure 6: Histogram of request size for example workload

correspond to each bin. Figure 6 shows a histogram of request size for the example workload for which each bin is 2KB wide. Notice that the sum of all bin values is equal to the number of I/Os measured.

The number of bins in a histogram determines its precision. If there are fewer bins than the number of unique values in the parameter's range, then the histogram does not present the distribution exactly. Figure 6 shows that three I/Os have request sizes of at most 2KB; however, we cannot determine precisely how many I/Os have 1KB request sizes and how many have 2KB request sizes. For operation type and request size, having one bin for each potential value is not problematic; however, for interarrival time and location, we must often limit the number of bins to make the histogram smaller than the workload trace itself.

Joint distribution: In addition to measuring the distribution of values for a single request parameter, we can also measure the empirical distribution of tuples

Table 3: Joint distribution of operation type and request size

Op. type	Request size							
	1KB	2KB	3KB	4KB	5KB	6KB	7KB	8KB
Read	1	0	0	1	0	0	0	2
Write	1	1	0	0	0	0	0	2

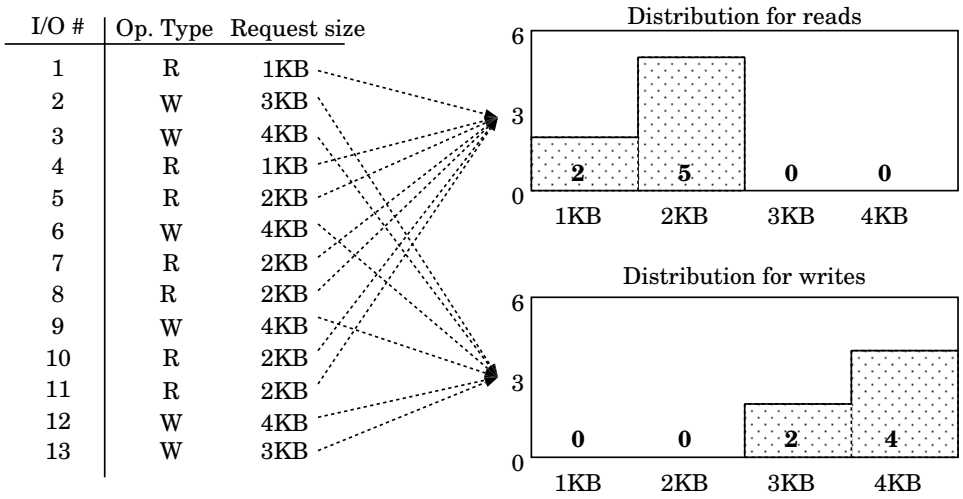

Figure 7: Conditional distribution of request size based on operation type

of I/O request parameters, called *joint distributions*. Table 3 shows the example workload's joint distribution of (operation type, request) pairs.

Conditional distribution: Instead of measuring a request parameter's distribution of values over all of a workload's requests, we can measure the distribution of values for only those I/Os that meet a specified condition. We call the parameter being measured the *dependent parameter*; we call the parameter (or parameters) on which the condition is based the *independent parameter*. For example, we can calculate separate distributions of request size for read requests and write requests — each distribution of request size is conditioned upon the operation type. In this case (shown in Figure 7), operation type is the independent parameter and request size is the dependent parameter. Similarly, we can partition the range of request sizes into states, then calculate separate distributions of request size based on the state of the previous request size. In this case, request size is both the dependent and independent

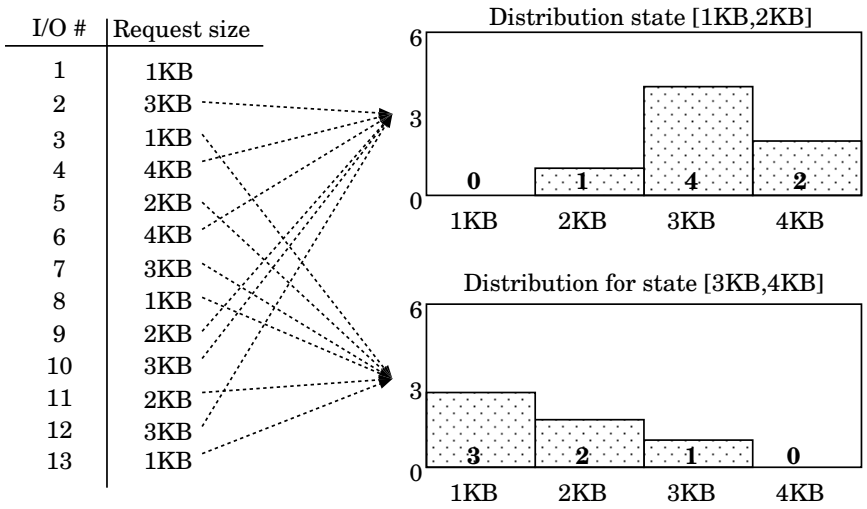

Figure 8: Conditional distribution of request size based on previous request size

parameter. Figure 8 shows an example with states [1KB, 2KB] and [3KB, 4KB].

We can define the states on which we base a conditional distribution in an arbitrary manner. One obvious technique is to assign each value in the independent parameter's range to a unique state. This technique works well for operation type because there are only two states (read and write); however, it is rarely used to study interarrival time or location because nearly every I/O may correspond to a unique state.

Unless we specify otherwise, we divide a parameter's range into states according to percentiles. For example, we partition the range of location values into four states as follows:

- State 0: locations below the 25th percentile
- State 1: locations between the 25th and 50th percentiles
- State 2: locations between the 50th and 75th percentiles
- State 3: locations above the 75th percentile

This method produces a set of states for which an equal number of I/Os correspond to each state. When applying this method to request size in the example workload,

we define two states as follows: [0, 7KB] and [8KB]. Each state contains exactly four of the eight I/Os. After testing several different techniques, we found this technique to be most useful.

As with empirical distributions, we can measure joint distributions (i.e., the dependent parameter can be a tuple of parameter values). Similarly, we can define the independent parameter to a tuple of parameter values comprising values for different parameters of the same I/O request (e.g., (operation type, location)), values from successive I/O requests (e.g., (previous location value, current location value)), or both.

When specifying a conditional distribution that uses a tuple for the independent parameter, one must be mindful of the total number of distributions because it grows multiplicatively with the number of states and dimension of the tuple. For example, using (location, request size) as an independent parameter, where location and request size each have 10 states, defines $10 \cdot 10 = 100$ distributions. Using the previous four interarrival times as an independent parameter with interarrival time divided into 10 states defines $10^4 = 10000$ distributions.

We found that, in practice, many useful attributes fit a single template based on the conditional distribution. This template requires four parameters:

1. the *independent parameter, i*: the request parameter(s)[1] on which the states are based;

2. the *dependent parameter, d*: the request parameter(s)[1] being measured;

3. the *history h*: the number of I/Os considered; and,

4. the *number of states, s,* into which the range of the independent parameter is divided.

[1]The dependent and independent parameters can be tuples of I/O request parameters, or even other attributes (such as jump distance or run count).

This template specifies a set of s^h distributions based on the h most recent independent parameters. In most cases, h refers to the current I/O and the $h-1$ previous I/Os; however, if the dependent and independent parameters are identical (e.g., distributions of request size based on the previous request size), h refers to the number of previous I/Os. (It does not make sense to define a distribution of request size based on the current request size.)

Henceforth, we will use CD(i, d, h, s) to specify a conditional distribution. For example: CD(operation type, location, 1, 2) specifies a conditional distribution in which operation type is the independent parameter (i), location is the dependent parameter (d), the states are based on only the current I/O ($h = 1$), and operation type has two states ($s = 2$) (read and write). Thus, this attribute presents separate location distributions for read requests and write requests. CD(interarrival time, interarrival time, 4, 10) specifies a conditional distribution that presents 10000 separate distributions of interarrival time based on the interarrival times of the previous four requests.

Our conditional distribution template also has two additional parameters for which we always use default values. First, we use only the aforementioned "percentile" method to define the s states. Second, the conditional distributions have a default number of bins based on the dependent parameter:

- The histogram for operation type always has two bins: read and write.
- The histogram for request size always has 128 bins: one for each legal request size.
- The histogram for interarrival time always has 1024 bins. We use a log histogram in which the bin widths grow exponentially as time increases.
- Finally, the histograms for location contain one bin for each unique location

Table 4: State transition matrix for operation type

History = 1

	Current state	
Prev. state	Read	Write
Read	.5	.5
Write	.333	.667

History = 2

	Current state	
Prev. states	Read	Write
Read, Read	.5	.5
Read, Write	0	1
Write, Read	0	1
Write, Write	1	0

value. Because our workloads have address spaces that range from a few gigabytes to hundreds of gigabytes, histograms of locations tend to be quite large. However, we found that using a histogram of location with a fixed number of bins produces very inaccurate synthetic workloads. Chapter 7.3 explores the tradeoff between the size of the location histograms and the accuracy of the resulting synthetic workload. In addition, we are researching new, more compact techniques for accurately representing a workload's distribution of location.

3.3.2 State transition matrix

A **state transition matrix** takes a partition of an I/O parameter's range into states and, for each pair of states (x, y), lists the probability that an I/O request with a value corresponding to state x is followed by a request with a value corresponding to state y. Table 4 shows the example workload's transition matrix of operation type. We see that the probability an I/O request is a read is 50% if the previous request was a read, and 33% if the previous request was a write.

We can define the states for a state transition matrix to be tuples of values and will specify the tuples using the same conventions we use to specify conditional distributions: h specifies the number of I/Os considered, and s specifies the number of regions into which we partition the independent parameter's range. Table 4 shows the example workload's transition matrix of operation type when $h = 2$.

Interarrival time with self-transition count: Given a division of the range of interarrival times into states, Interarrival time with self-transition count (IAT STC)

I/O #	Interarrival time	
1	.1	Sequence length 4
2	.2	
3	.1	
4	.3	
5	.07	Sequence length 3
6	.02	
7	.01	
8	.5	Sequence length 1
9	.003	Sequence length 3
10	.002	
11	.006	
12	.03	Sequence length 1

Prev. state	Current state [.001.01)	[.01,.1)	[.1,1)
[.001.01)	0	1	0
[.01,.1)	0	0	1
[.1,1)	1	.5	.5

Transition matrix
(Transitions counted between sequences only)

Figure 9: Interarrival time run count

measures the lengths of sequences of I/O requests whose interarrival times all fall within the same state. This attribute also measures the matrix of transitions between regions so that the synthetic workload can maintain the same transitions between sequences. Figure 9 provides an example.

Operation type with self-transition count: Operation type with self-transition count (Operation type STC) measures the lengths of sequences of I/O requests that are all reads and the lengths of sequences that are all writes, then presents separate distributions of sequence length for reads and writes. This attribute is similar to Interarrival time STC, except there are only two states; therefore, the transitions are always from read to write and write to read.

3.3.3 Jump distance

The **jump distance** attribute presents the empirically observed distribution of jump distances between adjacent I/Os. Jump distance can also serve as either a dependent or independent parameter for conditional distributions.

We found that, in practice, a synthetic workload that maintained a given distribution of jump distance was inaccurate unless it also maintained the target workload's

Table 5: Jump distance within state

Jump distances

Location	Size	State	JWDS	mJDWS
0KB	1KB	1	NA	NA
3KB	1KB	1	2	3
5KB	1KB	1	1	2
10KB	1KB	2	NA	NA
11KB	1KB	2	0	1
3KB	1KB	1	-3	-2
17KB	1KB	2	5	6
0KB	1KB	2	-4	-3
19KB	1KB	2	1	2
1KB	1KB	1	0	1

Transition matrix ($h = 1$)

	State 1	State 2
State 1	.4	.6
State 2	.75	.25

Transition matrix ($h = 2$)

	State 1	State 2
State (1,1)	.5	.5
State (1,2)	.67	.33
State (2,1)	0	1
State (2,2)	1	0

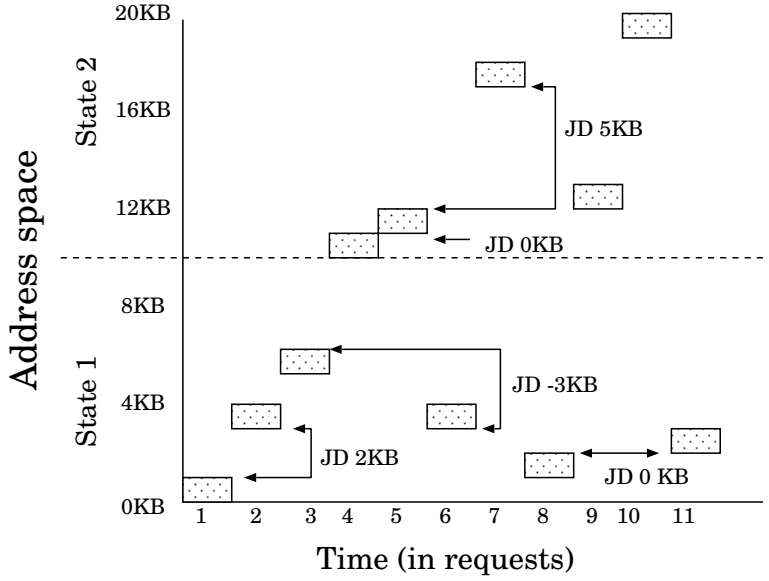

Figure 10: Jump distance within state

distribution of location. Therefore, when presenting a distribution of jump distance, we also present the distribution of location values.

Modified jump distance: The modified jump distance attribute presents the empirically observed distribution of modified jump distances between adjacent I/Os. Modified jump distance can serve as either a dependent or independent parameter for conditional distributions. As with jump distance, when presenting a distribution of modified jump distance, we also present the distribution of location.

Jump distance within state: The jump distance within state (JDWS) attribute

presents the distribution of jump distances between the beginning of the current request and the end of the most recent request corresponding to the same state as the current request. Unless specified otherwise, we always base the states on location. Table 5 and Figure 10 provide an example in which all request sizes are 1KB and the states are [0, 9KB] and [10KB, 29KB].

We present the distribution of jump distance for each state, the matrix of transitions between states, and the distribution of location. When referring to a jump distance within state attribute, we will give the history (h) and number of states (s). The history specifies how many previous locations we used to specify the states for the transition matrix. The amount of history affects the transition matrix only. A jump distance within state attribute with parameters ($h = 2$, $s = 4$) will have four distributions of jump distance; and, the transition matrix will have $4^2 = 16$ states.

Jump distance within state (RW): This attribute is similar to jump distance within state, except read requests and writes requests are analyzed separately (i.e., the states are (operation type, location) pairs). As with jump distance within state, s refers to the number of regions into which the address space is divided. Thus, a jump distance within state (RW) attribute with $s = 4$ will have four location regions but $4 * 2 = 8$ distributions of jump distance (four for reads and four for writes).

Because states are (operation type, location) pairs, the state transition matrix implicitly defines a Markov model of operation type. When $h > 1$ then the states for the transition matrix include the h most recent I/Os. Thus, when $h = 3$ and $s = 4$, there are $(4 * 2)^3 = 256$ states in the transition matrix. Should the user choose not to generate operation type using the implicit Markov model, she can set the *pol* ("preserve operation list") flag. See Appendix A for more details.

Modified jump distance within state and **Modified jump distance within state (RW):** These attributes are similar to jump distance within state and jump distance within state (RW), except they use modified jump distances (i.e., calculate

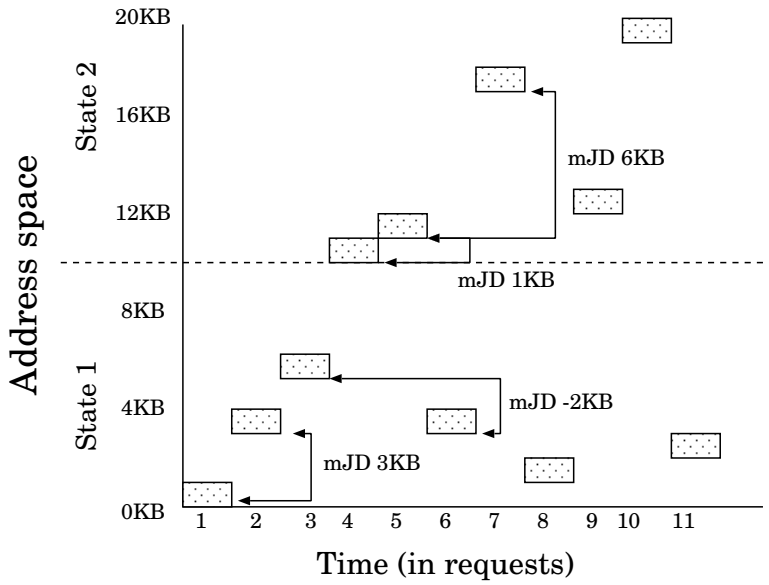

Figure 11: Modified jump distance within state

jump distance by subtracting the location values of two successive requests). Table 5 and Figure 11 provide an example. When presenting modified jump distance within state (mJDWS), we also present the appropriate distribution or distributions of location.

3.3.4 Run count

The **run count** attribute presents the empirically observed distribution of run counts (i.e., number of successive I/Os with zero jump distances). Notice a workload may contain fewer runs than I/Os because each run comprises several I/Os. As with jump distance, run count can serve as either a dependent or independent parameter for conditional distributions.

To generate a workload with a given distribution of run count, the generation technique must (1) choose a location for the heads of the runs, and (2) choose the run count for the current run. Therefore, when presenting a run count attribute, we also include an attribute that describes locations at the head of the runs (distribution, conditional distribution, jump distance, etc.).

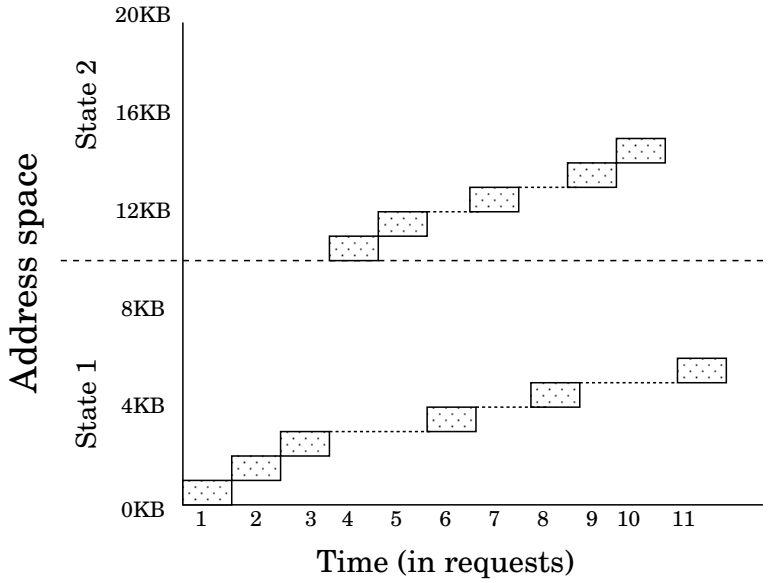

Figure 12: Run count within state

Modified run count: The modified run count attribute presents the empirically observed distribution of modified run counts. As with run count, when presenting a distribution of modified run counts, we also include an attribute that describes locations at the head of the runs.

Run count within state: The run count attribute considers only runs of strictly sequential I/Os. The run count within state (RCWS) analyzer considers the previous location in the same state when determining the length of a run. For example, consider the following sequence of locations: 0KB, 1KB, 2KB, 10KB, 11KB, 3KB, 12KB, 4KB, 13KB, 14KB, 5KB. Assume each request size is 1KB, and define states to be [0, 9KB] and [10KB, 19KB]. Run count within state defines two runs: 0KB, 1KB, 2KB, 3KB, 4KB, 5KB; and 10KB, 11KB, 12KB, 13KB, 14KB. Figure 12 highlights these two different runs. Notice that for run count within state to work as intended, the runs must lie entirely within different states.

The presentation of run count within state includes a separate histogram for each state and the state transition matrix. We specify history and states using the same conventions we used for jump distance within state. As with run count, we also

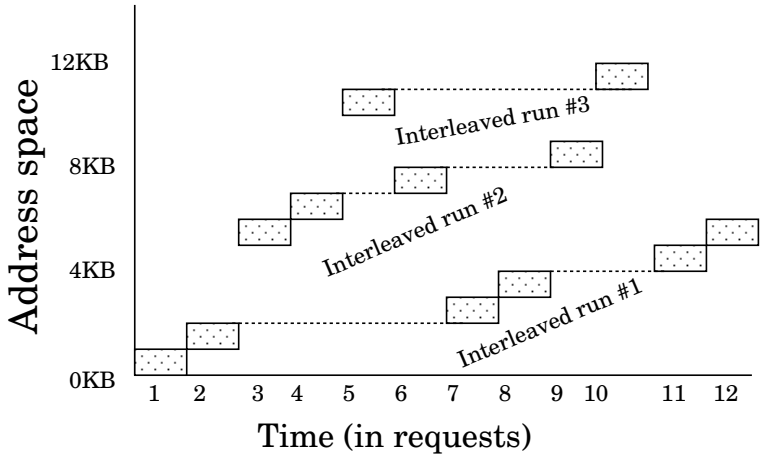

Figure 13: Interleaved runs

include an attribute that describes locations at the head of the runs.

Run count within state (RW): This attribute is like run count within state, except reads and writes are analyzed separately. Like jump distance within state (RW), run count within state (RW) implicitly contains a Markov model of operation type. When presenting run count within state (RW), we also present the distribution of read locations and the distribution of write locations at the head of runs.

Modified run count within state and **Modified run count within state (RW):** These attributes are like run count within state and run count within state (RW), except they present modified run counts instead of run counts.

Interleaved runs: We define an *interleaved run* (IR) to be a set of I/Os that form a run when considered apart from the rest of the workload. An interleaved run is like a run from run count within state; however, the user does not explicitly define the states. When analyzing a synthetic workload, if the current I/O does not continue any current interleaved run, a new run is created. The analyzer assigns each I/O to exactly one interleaved run. (In the case that an I/O can reasonably belong to more than one interleaved run, the analyzer assigns it to the most recently accessed interleaved run.)

Figure 13 shows several I/Os and the interleaved run to which they are assigned.

Notice also that I/O number 12 is assigned to interleaved run 1, even though I/O number 1 also has location 15 and was assigned to interleaved run 2.

For each interleaved run, this attribute reports the beginning location, number of I/Os, starting I/O number, and ending I/O number. In the degenerate case, each I/O forms a separate interleaved run; therefore, this attribute is most useful when studying workloads with a small number of large interleaved runs.

We can configure the attribute to consider interleaved runs with small gaps (jump distances between component I/Os that are greater than 0). We call this gap the "maximum gap."

3.3.5 Burstiness

The Distiller's library includes two attributes that quantify burstiness: the β-model, and IAT clustering.

β-model: The β model quantifies the amount of burstiness in a workload's arrival pattern using β, a parameter is related to the slope of the entropy plot [56]. (See Section 2.3.2.) We found that many of our test workloads are self-similar only when analyzed a few seconds at a time. Therefore, the implementation of the β-model in the Distiller's library allows the user to specify a window size. We then analyze each window separately, as if it were a separate trace. We also allow the user to specify the number of aggregation levels in the entropy plot by specifying the length of the smallest interval. For example, specifying an interval length of .01s and a window size of 5.12s creates 512 intervals per window and $\log_2 512 = 9$ possible aggregation levels. Finally, the generation technique requires an additional parameter called "sensitivity." This parameter specifies how many times the β-model's generation technique will recursively distribute the I/Os. Henceforth, when referring to the β model, we will also provide the three parameters used: window size, interval length, and sensitivity (e.g., β-model($ws = 5.12$, $il = .01$, $s = 10^{-5}$)).

IAT clustering: Hong and Madhyastha implemented a technique for choosing representative clusters of interarrival times [32, 33]. (See Section 2.3.2.) When using this attribute, the user must provide three parameters: the interval length (ws), the bin size used to calculate the burstiness (bs), and the desired percentage of intervals that will be cluster representatives (c).

Distribution clustering: We apply Hong and Madhyastha's interarrival time clustering technique to the workload's distribution of location values. Specifically, we divide the workload's address space into intervals (we experimented with interval lengths from 8MB to 128MB), and characterize each interval using the number of requests, aggregation ratio (the ratio of non-empty sectors to empty sectors), and entropy (using a similar technique to the β-model). As with IAT clustering, we then group the intervals into clusters using an agglomerative hierarchal clustering technique and choose one representative interval for each cluster.

This attribute allows us to reduce the amount of data required to precisely specify the distribution of location. Unfortunately, we completed this attribute and added it to the Distiller's library during the later stages of this thesis work; therefore, it was available only while performing the experiments presented in Chapter 7

3.3.6 Attribute groups

We organize our library of attributes by partitioning them into groups based on the I/O request parameters considered when computing the attribute's value. Formally, we define an *attribute group* to be a set of attributes whose values are calculated using the same set of I/O request parameters. For example, the {interarrival time} attribute group contains those attributes that measure only the interarrival times of different requests (distribution of interarrival time, β-model, etc.). The {operation type, location} attribute group contains those attributes that measure the relationship between requests' locations and operation types (modified JDWS, CD(operation type,

Table 6: Groups of candidate attributes

Attribute Group	Attribute
Any	Empirical distribution
	Implicit distribution
	Conditional distribution
{operation type}	Operation type STC
{interarrival time}	β-model
	IAT clustering
	Interarrival time STC
{location}	Modified jump distance
	Modified jump distance within state
	Modified run count
	Modified run count within state
	Interleaved runs
{location, request size}	Jump distance
	Jump distance within state
	Run count
	Run count within state
{operation type, location}	Modified jump distance within state (RW)
	Modified run count within state (RW)
{op. type, location, req. size}	Jump distance within state (RW)
	Run count within state (RW)

location, 1, 2), etc.). All non-empty subsets of {location, request size, operation type, request size} define fifteen attribute groups. By definition, each attribute is a member of exactly one attribute group.

Table 6 lists the attributes we have implemented and their attribute groups. Empirical and conditional distributions apply to all attribute groups. Table 6 does not explicitly list those attribute groups containing only empirical distributions and conditional distributions.

Attributes in the same group quantify the same qualitative workload properties. For example, {interarrival time} attributes all quantify the workload's burstiness (or lack thereof). Attributes in the {location} attribute group all quantify locality (spatial and/or temporal). We will see in Section 5.3 how the Distiller leverages this partitioning of attributes by qualitative workload properties.

3.3.7 Attribute size

An attribute's size is the number of bytes needed to represent its value. For example, we can represent a workload's read percentage (i.e., distribution of operation type) using only a few bytes (the size of one floating point value). Similarly, the size of a histogram of request size is approximately 1024 bytes — eight bytes for each of 128 1KB bins. In general, however, the precise size of an attribute depends on its configuration and the method (e.g., text or binary) used to represent the data.

The size of most attributes depends on their configuration. For example, the size of a distribution depends on the number of bins in the histogram used to represent the distribution. The size of a state transition matrix depends on the number of states. The size of the β-model depends on the number of windows analyzed.

The method used to represent the data also affects the attribute's size. For example, the precise size of a workload's read percentage or distribution of request size depends on the number of bits used to represent a floating-point value. Similarly, we could increase the size of these attributes by using an ASCII characters instead of a binary format.

Some attributes can have many different reasonable representations. For example, when presenting a histogram, we can list the number of items in each bin, or we can list the bin number and value for non-empty bins only. The latter representation is useful for presenting distributions of location that have many empty bins. In this case, the workload also affects the size of the attribute because different workloads have different sets of location values.

Determining a minimal representation of a given attribute is not practical. Therefore, for this thesis, we approximate the size of an attribute by compressing its representation using bzip2. When comparing the size of an attribute or set of attributes to the size of a workload trace, we also compress the workload trace using bzip2.

We now describe the approximate size of the attributes in our library. Unless

specified otherwise: b is defined be the number of bins used to present a distribution, s is the total number of states into which the independent parameter(s) is divided, h is the history (i.e., number of recent I/Os considered), and T is the number of I/Os in the workload trace.

- **Empirical distribution:** $O(b)$.
- **Conditional distribution:** $O(s^h b)$. However, the size could be much smaller if many of the defined distributions do not contain any I/Os.
- **State transition matrix:** $O(s^{2h})$.
- **Jump distance, modified jump distance:** $O(b)$.
- **JDWS, modified JDWS:** $O(sb) + O(s^{2h})$.
- **JDWS (RW), modified JDWS (RW):** $O(sb) + O(s^{2h})$.
- **Run Count, modified run count:** $O(b)$.
- **RCWS, modified RCWS:** $O(sb) + O(s^2)$.
- **RCWS (RW), modified RCWS (RW):** $O(sb) + O(s^h)$.
- **Operation type run count:** $O(b)$.
- **Interarrival time run count:** $O(sb) + O(s^{2h})$.
- **Interleaved runs:** $O(r)$, where r is the number of interleaved runs found.
- **β-model:** $O(w)$, where w is the number of windows chosen (i.e., number of separate intervals analyzed).
- **IAT clustering:** $O(T)$.

3.4 Evaluation metrics

This section discusses the metrics we use to compare the behavior of two workloads. Our goal is for the synthetic and target workloads to have similar behavior. We can consider many different behaviors (e.g., response time, throughput, power consumption); and can compare some behaviors with many different metrics. The Distiller's design is independent of the specific behavior and similarity measure chosen.

For this thesis, we study only one behavior: response time — the metric of primary interest to most researchers. Throughput is also important; however, because we are using an open workload model, the throughput of almost all synthetic workloads will be similar to that of the target workload. The most notable exception will be for those workloads that nearly saturate the storage system.

Section 3.4.1 defines four metrics with which we quantify differences in workload performance. These comparison metrics are used both "externally" by the user to evaluate the quality of the Distiller's answers and "internally" by the Distiller to automatically choose attributes. Section 3.4.2 discusses the various sources of differences in workload performance.

3.4.1 Evaluation criteria

We consider four metrics for comparing workload performance: (1) mean response time, (2) root-mean-square, (3) log area, and (4) a hybrid of log area and the mean response time. For each of these metrics, we define a *demerit figure*. The demerit figure is a normalized representation of the respective similarity metric. In each case, a larger demerit figure indicates less similarity between the workloads evaluated. Their prominence in the literature makes the mean response time and root-mean-square metrics *de-facto* standards [21, 24, 25, 32, 46, 47]. Despite their prominence, these metrics have limitations. The log area and hybrid metrics are our attempts to address these limitations.

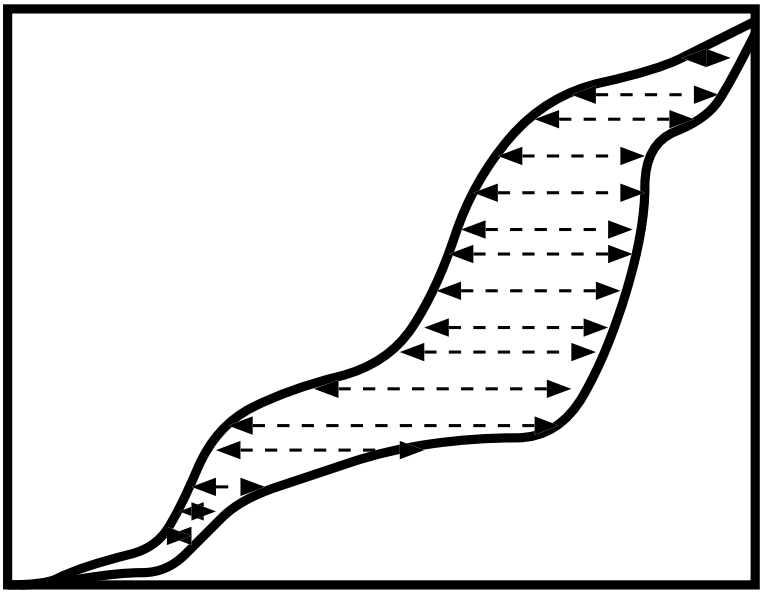

Figure 14: Calculation of RMS.

Mean response time: The mean response time (MRT) is the simplest response time metric. However, two workloads with very different behaviors can have the same mean response time. For example, one workload's I/Os may all have 10ms response times, while 99% of a second workload's I/Os may have .1 ms response times and 1% of the I/Os have 1s response times. The user will notice a delay in the second workload but not in the first. We will present the difference in mean response times (henceforth called the *mean response time demerit figure*) as the absolute value of the percent difference between the two response times.

$$100 * \frac{2 * |response\_time_1 - response\_time_2|}{(response\_time_1 + response\_time_2)}$$

RMS: Our primary metric is the root-mean-square (RMS) of the horizontal distance between the workloads' cumulative distribution functions (CDFs) of response time. RMS is similar to the average horizontal distance between the two CDFs; however, when computing the RMS, we square the horizontal distances to emphasize larger differences. Specifically, we take the horizontal difference between the two CDFs at each percentile, sum the squares of those differences, then take the square

root of that sum. Figure 14 illustrates the values that we square and sum when computing RMS. Appendix B contains pseudo-code for an algorithm that computes the RMS demerit of two distributions. The RMS demerit figure will be the RMS metric divided by the target workload's mean response time.

The RMS metric is popular because it emphasizes those differences that the user is most likely to notice. Because a CDF's top percentiles represent the number of high-latency I/Os, differences between the top percentiles of two CDFs will make the largest contribution to the RMS demerit. These high-latency I/Os are precisely the I/Os that the user is most likely to notice.

In many cases, it is useful to measure error from the "user's perspective"; however, this metric does not always reflect all differences in disk array behavior. For example, because the metric sums horizontal differences between CDFs, two workloads can have a small RMS demerit, but have very different cache behavior.

Log area: To address the limitations of the RMS metric, we developed a similar metric called *log area*. Instead of measuring the absolute horizontal distance, it measures the relative difference between distributions. This measurement is similar to the area between two CDFs when plotted on a log scale. The difference between 2ms and 4ms and the difference between 200 and 400 seconds affect the log area value equally. As a result, differences between both low-latency and high-latency I/Os contribute equally. Appendix B contains pseudo-code for an algorithm that computes the log area demerit of two distributions Section 7.1 examines the relative merits of the RMS and log area demerit figures.

The minimum log area value is 1.0. We will present the log area demerit figure as a percentage — specifically, $100 * (1 - \log\_area)$.

Hybrid: Some workloads can have small log area demerit, but a very large mean response time demerit. Having similar mean response times can be considered a necessary but not sufficient condition for declaring two workloads to be similar; therefore,

we define the *hybrid demerit figure* as the larger of the log area and mean response time demerits. Section 7.1 examines the merits of the hybrid demerit figure..

3.4.2 Error

A synthetic workload that perfectly represents the target workload trace has a demerit of 0%. However, due to various experimental errors, it is difficult to achieve identical performance. Ganger distinguishes between *synthesis error*, due to the different synthesis techniques, and *randomness error*, the error of a single synthesis technique using different random seeds [21]. When issuing I/O requests to a real storage system (as opposed to a simulator) we may also experience *replay error*: the experimental error due to non-determinism in the disk array and host operating system.

We used the technique presented in [21] to compute replay and randomness error. To compute replay error, we issue the workload to the storage system under test ten times. We then combine the CDF of response time for each replay into a single CDF representing the "average" CDF over all replays. Finally, we compute the demerit for each replay relative to the "average" CDF. The replay error is the mean of these demerits.

A workload's replay error represents the amount of "noise" in the measurement of a workload's quality. Therefore, the replay error limits the degree of accuracy we can claim for a synthetic workload (without repeating evaluations for statistical confidence). For example, given only a single evaluation, we cannot claim a workload has only a 3% error if the measurement system has a 5% margin of error. Our experiments indicate that replay error is typically around 3%, but can be as high as 14% for bursty workloads.

Our technique for computing randomness error is similar, except instead of issuing the same workload to the storage system under test, we generate ten workloads using ten different random seeds. As with the replay error, a synthetic workload's

randomness error can also serve as a lower bound on the measurement of its quality. However, we must be careful not to use the randomness error of a single synthetic workload as the Distiller's quality goal because one workload's high randomness error may indicate a poorly designed synthetic workload instead of a fundamental limit in our ability to evaluate synthetic workload quality. On the other hand, it is reasonable to consider the randomness errors of many synthetic workloads when setting a quality goal for the Distiller, assuming that most workloads considered are well designed. Our experiments indicate that the randomness error for synthetic workloads tends to be around 4%; however, synthetic workload specified using only one or two attributes can have randomness errors over 10%.

Given the observed replay and randomness errors, we initially set a quality goal of 10% error for the Distiller. Specifically, we configure the Distiller to terminate when it can specify a synthetic workload with a demerit of at most 10%. We will use this goal when evaluating the Distiller in Chapter 6.

The challenge with using the "percentage-based" demerit figures presented in section 3.4.1 is that the people's intuitive mapping of percent error to quality may not apply to our demerit figures. For example, unless told otherwise, most people will assume that a synthetic workload with a demerit of 20% or more is of marginal quality. However, it is not obvious whether a synthetic workload with a RMS or log area demerit of 20% is of good, bad, or marginal quality. The true measure of a synthetic workload's quality is whether a researcher can use it in place of a trace of a target workload for a storage system evaluation. Chapter 8 examines how workload's quality as measured by the MRT, RMS, log area, and hybrid demerit figures, relates to its usefulness.

CHAPTER 4

EXPERIMENTAL ENVIRONMENT

The Distiller finds the important performance-related attributes with respect to a specific workload and storage system. Sections 4.1 and 4.2 describe the storage systems and workloads we use in Chapters 6 through 8 to evaluate the Distiller. In addition, Section 4.3 discusses the eight existing software packages the Distiller uses.

4.1 Storage systems

We generated the experiments in this dissertation using either the Pantheon disk array simulator [57], the HP FC-60 disk array [2, 29, 36], or the HP FC-30 disk array [1, 28]. We used Pantheon to obtain most of our results because Pantheon allows us to study the largest workloads, is highly configurable, and is time efficient (we can run many simulations in parallel). Unfortunately, the developers have not yet rigorously validated Pantheon. The FC-30 and FC-60 allow us to evaluate the Distiller using real storage system hardware. However, these disk arrays can be used only in special situations: The FC-30 is very small and can handle only small or partial workloads. The FC-60 is larger, but rarely available. The remainder of this section provides the details of each storage system.

Pantheon: Pantheon simulates disk arrays comprising several disks connected to one or more controllers by parallel SCSI buses [57]. The simulated controllers have large NVRAM caches. This general architecture is similar to both the FC-60 and the FC-30. In addition, Pantheon provides many configuration parameters including:

- number and type of component disks,
- cache size,

Table 7: Comparison of FC-30 and FC-60

	FC-30	FC-60
Total size	120GB	.5TB
Number of disks	30	30
Cache size	60MB	512MB
Write behavior	Write-back	Write-back
Controllers	2	2
Connection	Fibre Channel	Fibre Channel
Separate R/W cache	Yes	No

- prefetch length,
- high- and low- water marks (points at which the cache begins de-staging dirty data),
- RAID stripe unit size, and
- speed of data connections.

Table 9 shows the Pantheon configuration we used to study each workload.

FC-60: The FC-60 disk array is populated with thirty 17 GB disks, spread uniformly across six disk enclosures, for a total of 0.5 TB of storage. It is configured with five six-disk RAID5 LUs, each with a 16 KB stripe unit size. The 512MB disk array cache uses a write-back management policy backed with non-volatile RAM. The FC-60 considers writes complete once it places the data in the cache, but commits the data to the disk media at a later time (a process called "de-staging"). Thus, from the perspective of the user, most writes appear as cache hits (i.e., almost "free").

FC-30: The FC-30 contains thirty 4GB disks (for a total of 120GB of storage) and two disk controllers with a total of 60MB of NVRAM. As with the FC-60, this disk array uses a write-back cache management policy backed with non-volatile RAM.

The user can specify the percentage of the cache to be used for reads and the percentage to be used for writes. The read cache stores only read data; and the write cache stores only write data. We configured the FC-30 with a 40MB write cache and

a 20MB read cache. In contrast, the FC-60's cache is not divided into a read and write caches. Table 7 compares the FC-30 with the FC-60.

4.2 Workloads

We obtained traces of four different workloads: an Email server (OpenMail), an on-line transaction database (OLTP), a decision support database (DSS), and a file server. The file server workload's characteristics are not static over time; therefore, it is not suitable for analysis by the Distiller. This section discusses the high-level characteristics of the three remaining production workloads. Table 8 lists the characteristics of the specific target we used in this dissertation. To limit the running time and storage needs of the Distiller, our target workloads are shorter than the full workload trace. The data presented in Table 8 applies to the target workload used. The data presented below applies to the entire trace collected.

For the OpenMail and DSS workloads, we configured Pantheon to use as few resources as possible without overwhelming the simulated storage system. The traces of the OLTP workload include a description of the storage system on which the workload was collected. Therefore, we configured Pantheon to match that storage system as closely as possible. Table 9 lists the Pantheon configuration we used to study each workload.

OpenMail: This one-hour trace of the OpenMail e-mail server contains 1289000 I/Os for a mean request rate of 358 I/Os per second. This trace comprises 22 LUs. Eight of the LUs have an address space of approximately 2.7GB; eleven have an address space of approximately 62GB. These LUs have between 50,000 and 80,000 I/Os each. The remaining three LUs have only a few hundred I/Os each.

The entire OpenMail workload is too large for the FC-30 and the FC-60. Therefore, to study this workload using these storage systems, we chose two sets of LUs of a more manageable size. The first partial workload, OM (2GBLU), comprises four

Table 8: Summary of workloads

Workload parameters							
Workload		I/Os	Throughput	Read pct.	Arrival rate (I/O/sec)	LUs	Length (sec)
OM	(All)	649616	2.33 MB/s	28%	361	22	1800
OM	(2GBLU)	78947	672 KB/s	90%	88	4	900
OM	(62GBLU)	45442	285 KB/s	27%	50	3	900
OM	(Sample)	19769	167 KB/s	29%	22	1	900
OLTP	(All)	1098290	1.35 MB/s	57%	610	37	1800
OLTP	(Log)	24402	141 KB/s	1%	27	1	900
OLTP	(Data)	19808	44 KB/s	68%	22	1	900
DSS	(All)	8373997	230 MB/s	83%	2326	80	3600
DSS	(1LU)	44777	6.21 MB/s	100%	50	1	900
DSS	(4LU)	332394	32.14 MB/s	100%	185	4	1800

Table 9: Summary of Pantheon configurations

Workload		Disks	Buses	RAID groups	Disk size	Disk type	Cache size	Bus rate
OM	(All)	180	4	45	9 GB	Seagate Cheetah 10K rpm	1 GB	40 MB/s
OLTP	(All)	80	2	40	1 GB	Wolverine III (HPC2490A)	256 MB	40 MB/s
DSS	(4LU)	16	4	4	9 GB	Seagate Cheetah 10K rpm	256 MB	100 MB/s

of the eight 2GB LUs. (Using more than 4 LUs overwhelms the FC-30). The second workload, OM (62GBLU), comprises three of the 62GB LUs. Although the address space of each LU is 62GB, there are no requests between 12GB and 50GB, and each LU only has one 2GB cluster of data between 50GB and 62GB. Therefore, we are able to map these three LUs onto the FC-30's 60GB of available storage. To evaluate the entire workload, OM (All), we used Pantheon to simulate a disk array with 180 9GB Seagate disks arranged into 45 18GB LUs. The simulated disk array has a cache size of 1GB. To conserve time and storage resources, we study only the first 1800 seconds of this trace. We use the workload OM (sample) in Chapter 5 for demonstration purposes.

The OpenMail workload has two different address regions: The lower portion of each LU's address range appears to correspond to OpenMail's index. These I/Os are mostly writes and exhibit a high degree of temporal locality. The upper region of each LU's address range appears to correspond to message data. These I/Os are mostly reads and exhibit a very low degree of spatial and temporal locality.

OLTP: This 1994 online transaction processing (OLTP) trace measures HP's Client/Server database application running a TPC-C-like workload at about 1150 transactions per minute on a 100-warehouse database. The entire workload comprises 4257935 I/O over 7900 seconds with a throughput of 1.20 MB/s.

We configure Pantheon to simulate a disk array with 80 1GB Wolverine disks arranged into 40 1GB LUs. To conserve time and storage resources, we study only the first 1800 seconds of this workload when using Pantheon. The entire workload is small enough to fit on the FC-30; however, the workload's 538 I/Os per second quickly overwhelm the array. Therefore, to study this workload on the FC-30, we examined the LUs individually. Table 8 shows several representative LUs.

Decision Support: This decision support system (DSS) trace was collected on an audited TPC-H system running the throughput test (multiple simultaneous queries)

on a 300 GB scale factor data set. This workload comprises 8374000 I/Os over 80 LUs. Most LUs are read-only; nearly all I/Os on these LUs are 128 KB in size. Several LUs, however, contain mostly writes. Each of the simultaneous queries generates a series of sequential I/Os. Visual inspection of the read-only LUs shows that many independent sequential streams are interleaved together.

Repeatedly analyzing, synthesizing, and simulating the entire DSS workload requires considerable computational and storage resources. Therefore, we have defined two smaller workloads. For study with Pantheon, we have chosen the four busiest LUs (4LU). The FC-30 cannot sustain the request rate of these four LUs, so we analyze only the busiest LU (1LU) when using the FC-30.

4.3 Software

The Distiller utilizes eight software packages and libraries. HP Labs developed some packages prior to this research; we developed others as part of this research.

1. **midaemon:** To collect the traces described in Section 4.2, we use the Measurement Interface Daemon (midaemon) kernel measurement system, a commercial product that is part of the standard HP-UX MeasureWare performance evaluation suite [30]. The midaemon collects the name, parameters, issue time, and completion time of every system call — including all I/O calls. From this data, we can extract the four parameters necessary for the workload traces (arrival time, operation type, request size and request location) as well as the information necessary to calculate the response time of individual I/O requests (completion time and arrival time).

2. **Lintel:** Lintel is the C++ library on which many Storage Systems Department tools are built [27]. Lintel includes many classes and modules for the implementation of workload analysis and storage management software. The tools the Distiller uses utilize the following Lintel features:

- assertion routines that provide error checking and debugging information,
- classes to collect and present histogram data,
- classes to generate random numbers, and
- code to provide a Tcl-language interface to tools with complex run-time configurations [23].

3. **srtLite:** The *srtLite* library contains C++ classes for reading and writing files in *SRT* format. SRT is a platform-independent, binary format used to store workload traces. It defines the byte-order and width of the fields used to describe the trace. In addition to operation type, arrival time, request size, and location (presented using a (deviceID, offset) pair), the SRT format also maintains:

   - **start time:** the time at which the operating system issues the request to the disk array;
   - **completion time:** the time at which the request completes;
   - **logical volume:** the logical volume that contains the data;
   - **queue length:** the number of outstanding requests;
   - **PID:** the process ID of the calling process;
   - **flags:** flags that denote whether the request is synchronous or asynchronous, and what type of file system information the request represents (data, cylinder group, directory, i-node, indirect i-node, fifo, etc.); and
   - **valid bits:** bits indicating the validity of each field.

   Not all traces contain data for each field. Our traces are guaranteed to contain only the four parameters necessary for our workload model.

4. **Rome:** We store workload characteristics and storage system descriptions in plain-text using a syntax called Rome [58]. Rome organizes data into objects

called *atoms*. Each atom is either a single datum (such as a double or a string) or a list of other atoms. An atom paired with a name and, optionally, a type is called an *attribute-value*. We use lists of attribute-values to build compound objects such as histograms and transition matrices. The Rome library includes generators and parsers in both C++ and perl.

5. **Rubicon:** Rubicon, our workload analysis tool, takes a workload trace and an attribute list as input and produces a text file in Rome format containing the attribute-values that characterize the workload [54].

6. **GenerateSRT:** Our workload generation tool, *GenerateSRT*, takes a workload characterization produced by Rubicon as input and generates a file (in SRT format) containing a synthetic workload matching that characterization. (See appendix A for more details.) We developed GenerateSRT concurrently with the Distiller.

7. **Buttress:** When studying real disk arrays, we use a tool called *Buttress* to issue a workload to a storage device. While we are running Buttress, we also run the midaemon trace collection tool to collect a trace of the synthetic workload for off-line performance analysis.

8. **Pantheon:** When simulating a storage system, we use the Pantheon disk array simulator. To use Pantheon, we pass the desired workload (in SRT format) as a parameter to Pantheon. Pantheon provides as output a set of statistics, including a distribution of response time latency.

Lintel, srtLite, Rome, and Rubicon are currently available on a limited basis to those in the academic community. Buttress and Pantheon contain trade secrets; however, HP Labs makes versions of these tools without the trade secrets available on a limited basis to those in the academic community. MeasureWare is a commercial

product sold by HP.

CHAPTER 5

DESIGN AND IMPLEMENTATION

In this section, we present our iterative approach for determining the attributes that are necessary for synthesizing a representative I/O workload. This approach is embodied in a tool we call the *Distiller*. The Distiller takes as input a *target* workload trace and a library of attributes. It then automatically searches for a subset of the library that effectively describes the target workload's performance-affecting properties. We first provide an overview of the Distiller's iterative approach; then, we discuss each step in detail.

5.1 Overview

Our approach to choosing attributes is to iteratively build a list of "key" attributes. During each iteration, the Distiller identifies one additional key attribute, adds it to the list, then tests the representativeness of the synthetic workload specified by that *intermediate attribute list* (the list of key attributes chosen thus far during the Distiller's execution). This iterative loop (shown in Figure 15) continues until either (1) the difference between the performance of the *intermediate synthetic workload* (specified by the intermediate attribute list) and the target workload is below some user-specified threshold, or (2) the Distiller determines that it cannot specify a more representative synthetic workload by adding a small number of attributes from the library.

The Distiller evaluates how well an intermediate attribute list captures the target workload's performance-affecting properties by (1) generating an intermediate synthetic workload with the same values for the chosen attributes, (2) issuing both the

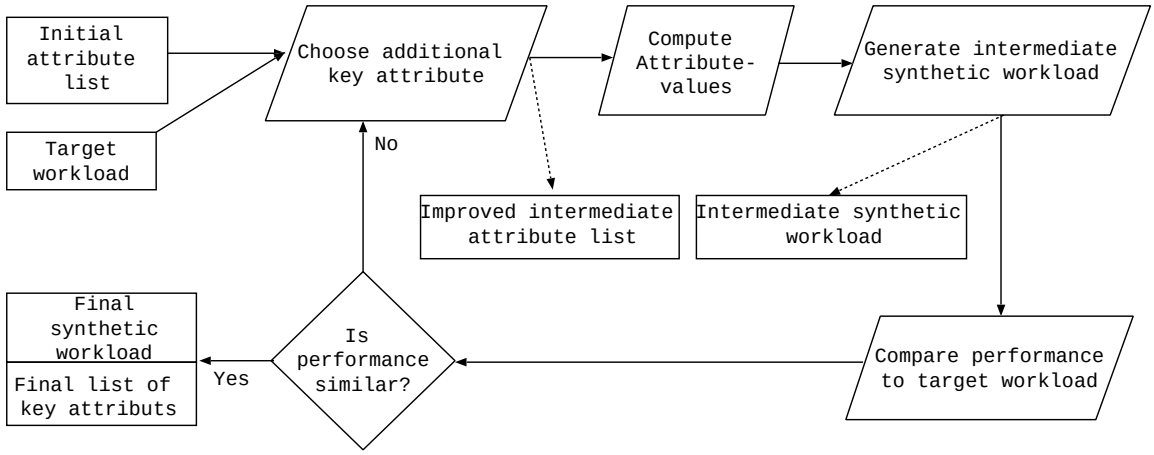

Figure 15: Distiller's iterative loop (detailed version)

intermediate synthetic workload and the target workload to the storage system under test, then (3) comparing the CDFs of response time for the workloads. If the two workloads have very different CDFs (as quantified using one of the demerit figures presented in Section 3.4.1), then we know that there is some important property that is not described by the intermediate attribute list. If the workloads have similar performance, then we argue that the chosen attributes capture the important performance-related properties.

Obtaining the distribution of response time for a workload takes several minutes. To keep the Distiller's running time reasonable, we limit the number of synthetic workloads the Distiller builds and evaluates by partitioning the library of attributes into groups and investigating only those groups that contain at least one "key" attribute. Section 5.3 defines attribute groups and explains how their use reduces the Distiller's running time; and Section 5.4 explains how we search an attribute group for key attributes.

Running Email example: To make the details of the Distiller's algorithm more concrete, we will use a running example to illustrate the Distiller's operation on a real production workload. Specifically, we will be using the workload OpenMail (sample) summarized in Table 8. For simplicity, this workload comprises only one LU. For this

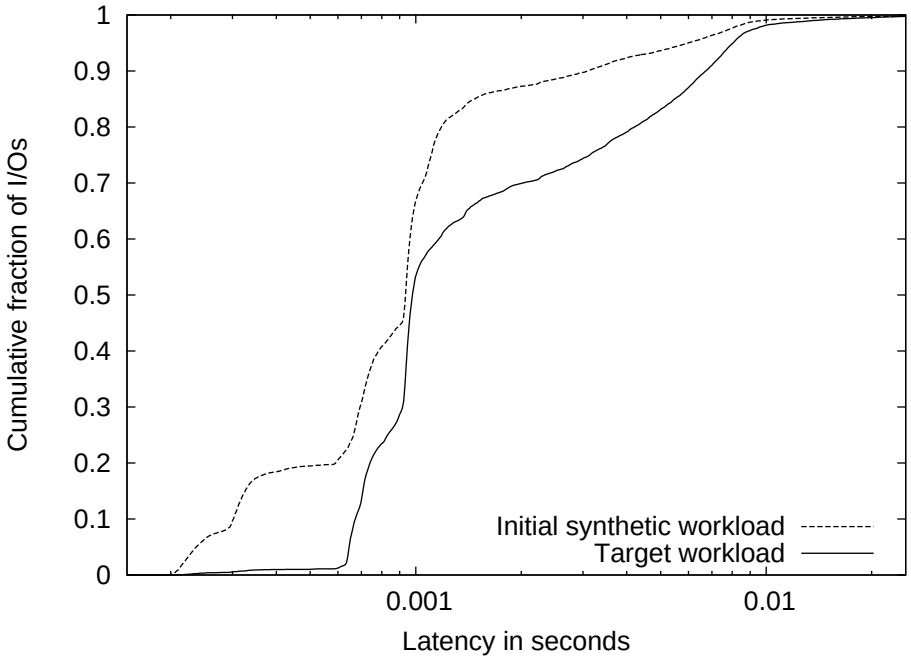

Figure 16: Initial synthetic workload is inaccurate

example, we will use the RMS demerit figure with a threshold of 12%.

5.2 Initial attribute list

The Distiller's first step is to generate an initial synthetic workload that maintains the empirical distributions for the four I/O request parameters. The simplest method of generating a synthetic workload (given our workload model) is to choose a value for each I/O request parameter randomly from some distribution. We use explicit empirical distributions because implicit distributions (e.g., normal or Poisson) have been shown to be inaccurate [21].

Running Email example: Figure 16 shows the response time distributions for the initial synthetic workload and the target OpenMail workload. The resulting RMS demerit is 65%. Note the log scale on the x-axis. The RMS demerit is larger than the threshold of 12%; therefore, the Distiller searches for additional key attributes.

5.3 Choosing an attribute group

In this section, we see how the use of attribute groups reduces the Distiller's running time. Evaluating an individual attribute requires generating an exploratory workload and obtaining its distribution of response time. When using the FC-30 or FC-60, we must wait 15 minutes to 1 hour for Buttress to issue the entire trace to the disk array in real time. Pantheon's 5 to 20 minute running time depends on the number of I/Os simulated. Our use of attribute groups reduces the Distiller's running time by providing a means for estimating the potential benefit of all the attributes in a group using the performance information from only a few exploratory workloads.

We define an *attribute group* to be a set of attributes whose values are calculated using the same set of I/O request parameters. For example, the {interarrival time} attribute group contains those attributes that measure only the interarrival times of different requests (distribution of interarrival time, β-model, etc.). The {operation type, location} attribute group contains those attributes that measure the relationship between requests' locations and operation types (modified JDWS, CD(operation type, location, 1, 2), etc.). All non-empty subsets of {location, request size, operation type, request size} define fifteen attribute groups. By definition, each attribute is a member of exactly one attribute group. Table 6 in Section 3.3.6 lists the attributes we have implemented and their attribute groups. (See Section 3.3 for detailed descriptions of the attributes.)

To determine whether an attribute-group contains any key attributes, we compare the performance of two *exploratory* synthetic workloads: The first preserves almost none of the attributes in the group under test. The second approximately preserves every attribute in that group. The difference in performance of the two exploratory workloads is an estimate of the importance of the attributes in the group. Should there be little or no difference in performance, we assume that we need not evaluate individual attributes in that group. However, if performance of the two exploratory

Table 10: Example of the subtractive method for {request size}

Target I/O workload				Subtractive workload			
Time	**Location**	**Op**	**Size**	**Time**	**Location**	**Op**	**Size**
0.050397	6805371	W	3072 (a)	0.050397	6805371	W	4096 (g)
0.762780	7075992	R	8192 (b)	0.762780	7075992	R	3072 (a)
0.789718	11463669	W	3072 (c)	0.789718	11463669	W	3072 (f)
0.792745	7051243	R	1024 (d)	0.792745	7051243	R	8192 (b)
0.793333	11460856	W	8192 (e)	0.793333	11460856	W	1024 (d)
0.808625	11463669	W	3072 (f)	0.808625	11463669	W	2048 (h)
0.808976	7049580	R	4096 (g)	0.808976	7049580	R	8192 (e)
0.809001	7050244	R	2048 (h)	0.809001	7050244	R	3072 (c)

workloads is dramatically different, we assume that the attribute group contains at least one key attribute.

The exploratory workloads we use to evaluate attribute groups depend on the number of request parameters that define the attribute group under test. We call the {operation type}, {location}, {interarrival time}, and {request size} attribute groups *single-parameter* attribute groups. We call those attribute groups defined using two parameters ({operation type, location}, {request size, interarrival time}, etc.) *two-parameter* attribute groups.

5.3.1 Single-parameter attribute groups

When evaluating single-parameter attribute groups, the first exploratory workload, called the *subtractive* workload, is identical to the target workload, except that we choose the values for the parameter under test independently at random from the empirical distribution of values in the target workload. Table 10 provides an example of the subtractive {request size} workload. The left half of the table shows the target workload; the right half shows the subtractive workload. By choosing the request size values randomly, we produce a synthetic workload that maintains almost no {request size} attributes. (We say "almost" because the distribution of request sizes is a {request size} attribute.)

The second exploratory workload, called the *rotated* workload, is identical to the

Table 11: Example of the rotated {request size} workload

Target I/O workload				Rotated workload			
Time	**Location**	**Op**	**Size**	**Time**	**Location**	**Op**	**Size**
0.050397	6805371	W	3072 (a)	0.050397	6805371	W	3072 (f)
0.762780	7075992	R	8192 (b)	0.762780	7075992	R	4096 (g)
0.789718	11463669	W	3072 (c)	0.789718	11463669	W	2048 (h)
0.792745	7051243	R	1024 (d)	0.792745	7051243	R	3072 (a)
0.793333	11460856	W	8192 (e)	0.793333	11460856	W	8192 (b)
0.808625	11463669	W	3072 (f)	0.808625	11463669	W	3072 (c)
0.808976	7049580	R	4096 (g)	0.808976	7049580	R	1024 (d)
0.809001	7050244	R	2048 (h)	0.809001	7050244	R	8192 (e)

target workload, except the values for the request parameter under test are "rotated" relative to the rest of the trace. To rotate a list L of values, we shift the list so that the order of the values is unchanged, but the value for request x is now the value for request $(x+t)$ mod $length(L)$ for some constant integer t. The right half of Table 11 shows the list of request sizes rotated by three requests. Currently, the Distiller is configured to rotate the parameters by $.5L$. Section 7.1 discusses the effects of the rotate amount on the Distiller's execution.

The rotated {request size} workload approximately maintains all {request size} attribute-values. The rotating of the last few values to the beginning of the trace will change some attribute-values slightly. For example, when computing the transition matrix for the workloads in Table 11, the rotated workload will have a transition from 2048 to 3072 (request values (h) and (a)), but the target workload will not.

Rotating the request sizes produces a synthetic workload that preserves every {request size} attribute-value, but preserves very few attribute-values from multi-parameter attribute groups involving request size ({request size, operation type}, {request size, location}, {request size, interarrival time}, etc.). Removing these attribute-values from multi-parameter attributes is important; if we did not, then we would be unable to determine whether any differences between the two exploratory workloads were due to {request size} attributes, or other multi-parameter attributes involving request size.

Table 12: Request size and operation type rotated together

Target I/O workload				Operation type and request size and rotated together			
Time	**Location**	**Op**	**Size**	**Time**	**Location**	**Op**	**Size**
0.050397	6805371	W (a)	3072 (a)	0.050397	6805371	W (f)	3072 (f)
0.762780	7075992	R (b)	8192 (b)	0.762780	7075992	R (g)	4096 (g)
0.789718	11463669	W (c)	3072 (c)	0.789718	11463669	R (h)	2048 (h)
0.792745	7051243	R (d)	1024 (d)	0.792745	7051243	W (a)	3072 (a)
0.793333	11460856	W (e)	8192 (e)	0.793333	11460856	R (b)	8192 (b)
0.808625	11463669	W (f)	3072 (f)	0.808625	11463669	W (c)	3072 (c)
0.808976	7049580	R (g)	4096 (g)	0.808976	7049580	R (d)	1024 (d)
0.809001	7050244	R (h)	2048 (h)	0.809001	7050244	W (e)	8192 (e)

Table 13: Request size and operation type rotated separately

Target I/O workload				Op. type and req. size rotated separately			
Time	**Location**	**Op**	**Size**	**Time**	**Location**	**Op**	**Size**
0.050397	6805371	W (a)	3072 (a)	0.050397	6805371	W (e)	3072 (f)
0.762780	7075992	R (b)	8192 (b)	0.762780	7075992	W (f)	4096 (g)
0.789718	11463669	W (c)	3072 (c)	0.789718	11463669	R (g)	2048 (h)
0.792745	7051243	R (d)	1024 (d)	0.792745	7051243	R (h)	3072 (a)
0.793333	11460856	W (e)	8192 (e)	0.793333	11460856	W (a)	8192 (b)
0.808625	11463669	W (f)	3072 (f)	0.808625	11463669	R (b)	3072 (c)
0.808976	7049580	R (g)	4096 (g)	0.808976	7049580	W (c)	1024 (d)
0.809001	7050244	R (h)	2048 (h)	0.809001	7050244	R (d)	8192 (e)

After constructing the subtractive and rotated exploratory workloads, the Distiller issues each to the storage system under test (which might be simulated), obtains the response time for each, then compares the two CDFs using one of the metrics discussed in Section 3.4.1. If the two workloads have a demerit above a user-specified threshold, then the Distiller searches the attribute group for a key attribute. Otherwise, the Distiller assumes that the attribute group under test contains no key attributes and that the empirical distribution is sufficient.

5.3.2 Two-parameter attribute groups

The exploratory workloads we use to evaluate two-parameter attribute groups are both rotated workloads. The first exploratory workload rotates the parameters under test together; the second exploratory workload rotates the two parameters by different

amounts. For example, to determine whether there are any key {operation type, request size} attributes, the Distiller generates an exploratory workload by rotating both request size and operation type $.25L$. The Distiller then generates an exploratory workload by rotating request size $.5L$ and rotating operation type $.25L$. Table 12 provides an example of operation type and request size rotated together; and Table 13 shows request size and operation type rotated separately.

Rotating request size and operation type together preserves the {request size, operation type} attributes, but does not preserve the attributes of any other multi-parameter attribute group involving request size or operation type. Rotating request size and operation type by different amounts then removes the {request size, operation type} attributes. The only difference between the two exploratory workloads is that the rotated together workload maintains {request size, operation type} attribute-values and the rotated apart workload does not. Consequently, the difference in performance between the two exploratory workloads estimates the effects of the {request size, operation type} attributes.

As with single-parameter attributes, after constructing the two exploratory workloads, the Distiller obtains the response time for each, then compares the two CDFs. If the two workloads have a demerit above a user-specified threshold, then the Distiller searches the attribute group for a key attribute. Otherwise, the Distiller assumes that the attribute group under test contains no key attributes.

5.3.3 Search order

The Distiller first tests each single-parameter attribute groups for the presence of key attributes. We refer to these iterations as *phase one* of the distillation process. After examining the single-parameter attribute groups, the Distiller tests two-parameter attribute groups in *phase two*. Finally, if necessary, the Distiller will search for important three-parameter and four-parameter attributes. However, we have yet to

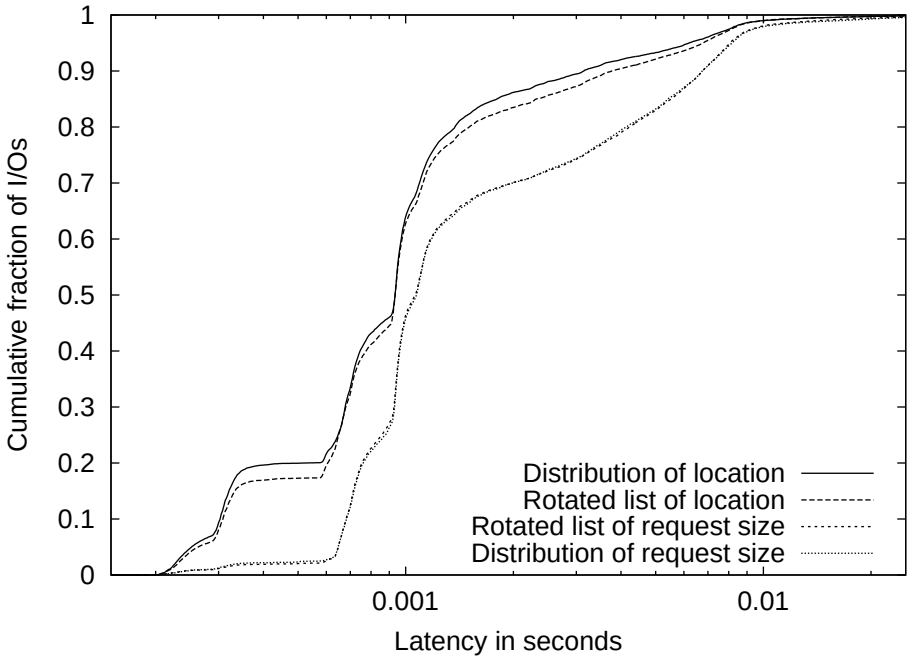

Figure 17: Testing for potential key {location} and {request size} attributes

encounter any workloads for which it is useful to proceed beyond phase two.

As an optimization, before testing an individual two-parameter attribute group involving the parameter p, the Distiller simultaneously tests all three two-parameter attribute groups involving p by comparing the rotated workload for p with the target workload. If the rotated and target workloads have a low demerit, then the Distiller assumes that there are no key attributes involving parameter p. The Distiller generated and evaluated the rotated workload during phase one, so this optimization does not add to the Distiller's running time.

For example, before testing any of the {operation type, request size}, {operation type, location}, or {operation type, interarrival time} attribute groups, the Distiller compares the performance of the rotated {operation type} workload to the target workload. If the demerit for the two workloads is small, then the Distiller assumes that there are no key {operation type, request size}, {operation type, location}, or {operation type, interarrival time} attributes.

Running Email example: The Distiller begins its exploration of attribute

groups by evaluating each single-parameter attribute group. Figure 17 shows two representative results for {request size} and {location}. The subtractive and rotated workloads for {request size} are similar, with an RMS demerit of only 8%; thus, no additional {request size} attributes (beyond the default empirical distribution) are necessary. Although the distributions for the subtractive and rotated {location} workloads look similar, the RMS demerit of 15% is above our threshold of 12%. Therefore, the Distiller will search for a key {location} attribute. The evaluations of the {interarrival time} and {operation type} attribute groups (not shown) indicate they probably do not contain any key attributes.

5.4 Choosing an attribute

Once the Distiller has identified an attribute group that potentially contains a key attribute, it must identify a specific key attribute. To do this search, the Distiller first orders all of the attributes in the attribute group according to the amount of data needed to store the corresponding attribute-value. (See Section 3.3.7.) It then evaluates each attribute in the attribute group beginning with the "smallest."

The Distiller uses two exploratory workloads to evaluate candidate attributes. The first exploratory synthetic workload uses the candidate attribute and preserves the list of original values for parameters not associated with the attribute under test. (In other words, we subtract all of the group's attribute-values and add back only the values for the attribute under test.) The second exploratory workload is the rotated workload for the group under test (i.e., it is identical to the target workload, except the parameter or parameters under test are rotated). In the case of a two-parameter attribute group, the Distiller uses the "rotated together" workload.

The Distiller then obtains the performance of the two exploratory workloads and

compares their distributions of latency. (Recall that the Distiller generated and obtained the performance information for the rotated exploratory workload when selecting the attribute group.) If the demerit for the two workloads is below a user-specified threshold, then the candidate attribute captures most of the important properties described by attributes in the group under test and the Distiller adds it to its intermediate attribute list. If the two workloads have a high demerit, the attribute is not helpful and the Distiller proceeds to evaluate other candidate attributes.

The user can configure the Distiller in several ways to optimize the search for a specific key attribute. First, setting the threshold to zero effectively changes the Distiller's first-fit search into a best-fit search. Also, instead of evaluating attributes in "size order", the user can specify the order in which the Distiller evaluates individual attributes. By specifying this order, the user can assure that the Distiller will evaluate the attributes she believes to be most useful first, thereby reducing the number of attributes evaluated (if she has chosen correctly).

If the Distiller evaluates every attribute in a group and finds none to be key attributes, then the library of attributes is insufficient. The user then has two options: (1) manually add more attributes to the library and re-start the Distiller, or (2) continue with the best available attribute from the library. For this dissertation, we always choose option (2).

Finally, generating a synthetic workload that simultaneously maintains the values of two different attributes from the same attribute group requires specialized code. In most cases, the generation techniques for maintaining individual attributes will interfere with each other when run simultaneously. To avoid this difficulty, the Distiller chooses only one key attribute per attribute group. Should an attribute group contain two or more key attributes, the user must combine the patterns measured into a single attribute and generation technique.

Running Email example: Recall from our earlier example that the Distiller

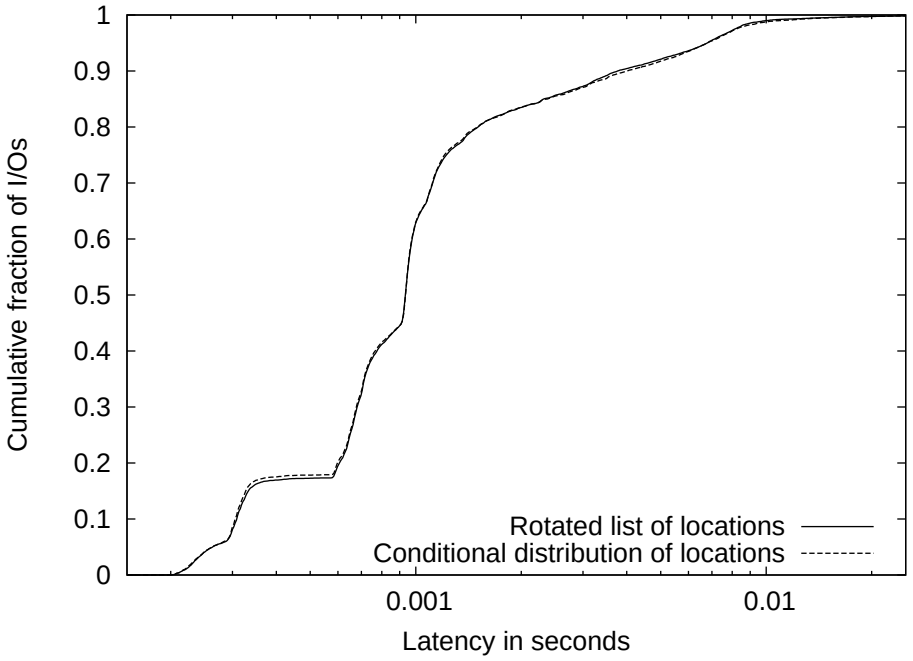

Figure 18: Conditional distribution of location closely matches rotated {location} workload

identified the {location} attribute group as potentially containing a key attribute. While exploring this attribute group, the Distiller evaluates a conditional distribution of location — CD(location, location, 1, 100). Figure 18 shows that the resulting synthetic workload behaves very much like a workload with the rotated sequence of location values. Therefore, the Distiller adds the conditional distribution of locations to the intermediate attribute list. No other single-parameter attributes contain any key attributes.

5.5 Completion of Email example

After the Distiller identifies a new key attribute, it evaluates the intermediate synthetic workload specified by the improved intermediate attribute list. If the new intermediate workload is sufficiently representative, the iterative process concludes. Otherwise, the Distiller continues its loop of evaluating attribute groups and candidate attributes.

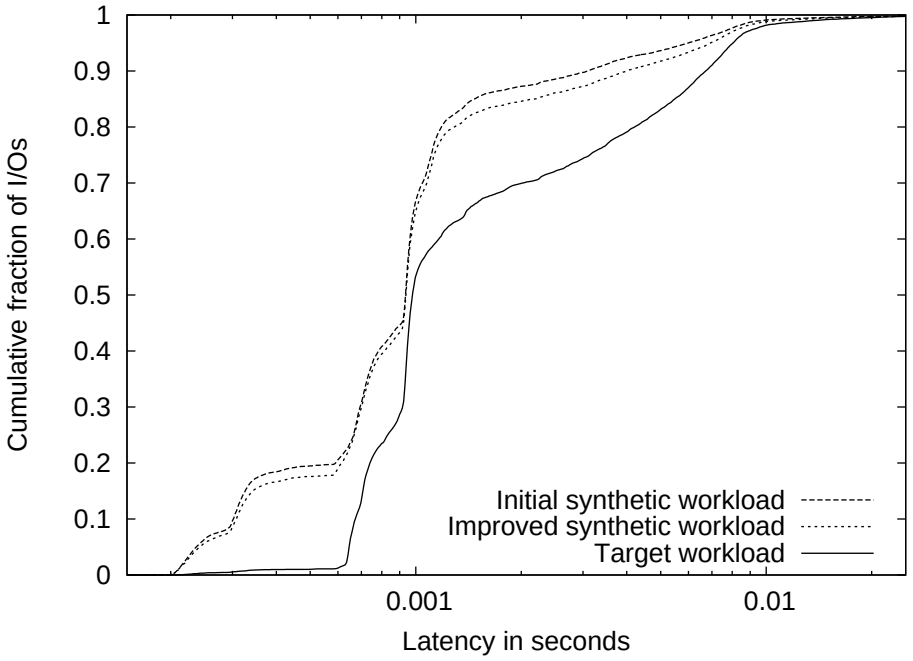

Figure 19: Evaluation of improved synthetic workload containing conditional distribution for {location}.

Running Email example: Figure 19 shows the evaluation of the intermediate attribute list containing the conditional distribution of location. Because the RMS demerit (54%) is still well above the desired threshold, the Distiller continues.

After addressing each single-parameter attribute group, the Distiller addresses the two-parameter attribute groups. Recall that the Distiller begins this phase by comparing the single-parameter rotated workload for each I/O parameter p to the original workload trace to evaluate the possibility of there being any key $\{p,\ x\}$ attributes (where x is any other I/O parameter).

Figure 20 illustrates this process. We see little difference in performance between the rotated request size workload and the target workload. Likewise, we see little difference in performance between the rotated interarrival time workload and target workload. However, the performance of the rotated workloads for operation type and location differ significantly (RMS demerits of 50% and 60%) from that of the target workload. Therefore, we conclude that there are some key {operation type, y}

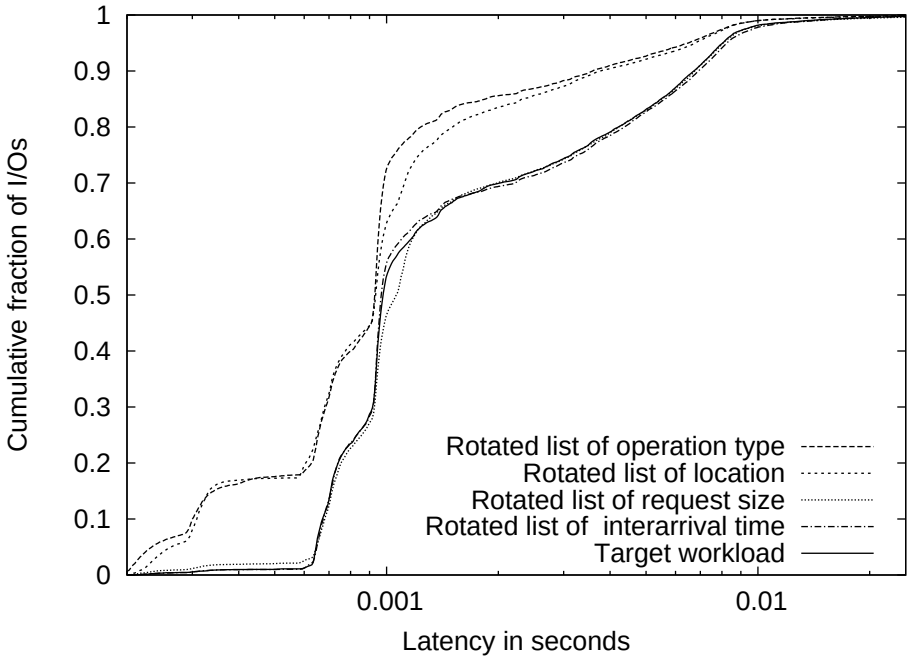

Figure 20: Testing for potential key two-parameter attributes

attributes and some key {location, z} attributes.

The Distiller next chooses specific two-parameter attribute groups to explore by comparing the "rotated together" and the "rotated apart" exploratory workloads, as described in Section 5.3. In the case of operation type, the Distiller evaluates the potential contribution of the {operation type, location}, {operation type, request size}, and {operation type, interarrival time} attribute groups. The high RMS demerit (56%) in Figure 21 indicates that there is probably a key {operation type, location} attribute. Other experiments (not shown) indicate that there are probably not any {operation type, request size} or {operation type, interarrival time} attributes.

When the Distiller has identified a two-parameter attribute group (in this case, {operation type, location}), it evaluates the candidate attributes as described in Section 5.4. For our example, the Distiller first evaluates a conditional distribution of location dependent on operation type. The resulting synthetic workload has similar behavior to the workload in which the Distiller rotated operation type and location together. Therefore, the Distiller adds this attribute to the intermediate attribute

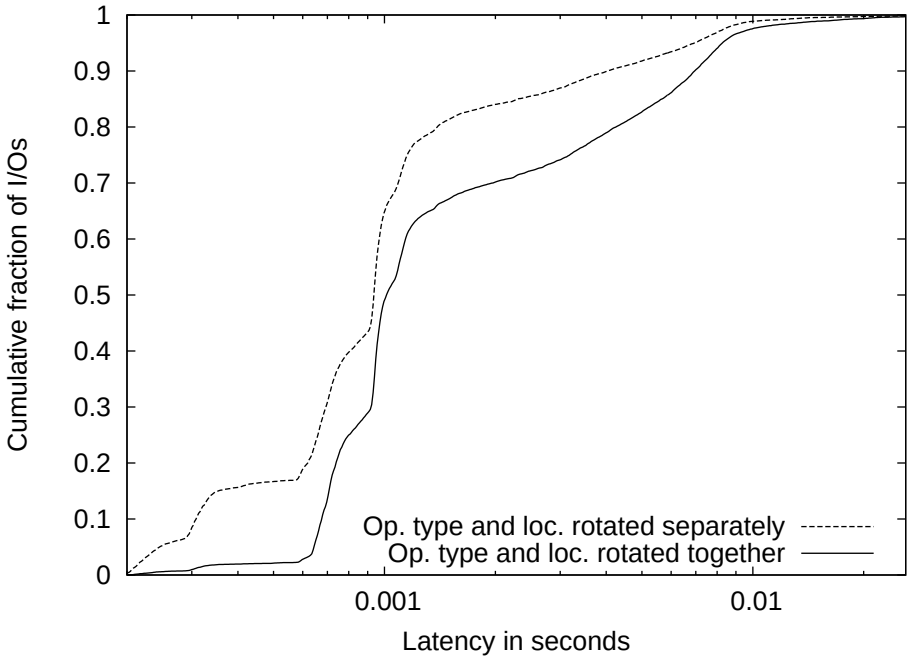

Figure 21: Testing for potential key {operation type, location} attributes

list.

Figure 22 shows the evaluation of the intermediate synthetic workload. Because the RMS demerit (8%) is below the desired threshold, the Distiller terminates. The final attribute list includes the default distributions for operation type, interarrival time, and request size; and a conditional distribution of location based on operation type — CD(operation type, location, 1, 2).

5.6 Limitations

The Distiller currently has three main limitations: (1) the synthetic workloads have randomness errors, (2) the target workload's characteristics are not static over time, and (3) the evaluation techniques for attribute groups only allow us to estimate whether an attribute group contains a key attribute.

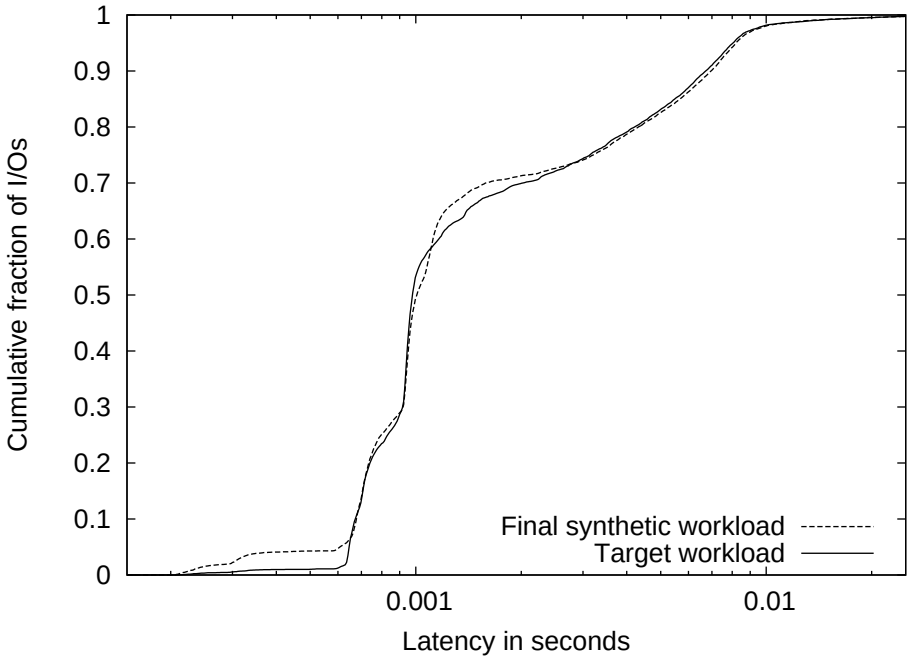

Figure 22: Evaluation of final synthetic OpenMail workload

5.6.1 Randomness error

The Distiller generates workloads randomly. The random seed used produces small difference between synthetic workloads with the same specification. If these differences have a large enough effect on performance, the attributes the Distiller chooses will depend on the seed used to initialize the random number generator when constructing exploratory workloads.

In general, we expect that attributes with more information (i.e., "larger" attributes) will have smaller randomness errors because they allow less variation between synthetic workloads. Chapter 6 will show that the final synthetic workloads have small randomness errors. Chapter 7 will show that simple synthetic workloads, such as the exploratory workloads constructed during phase one, are more seriously affected by randomness error.

One technique for reducing the effects of randomness error is to generate several synthetic workloads using different random seeds, issue each workload to a storage

system, then compute the "mean" distribution of response time (using the technique discussed in Section 3.4.2). Time and space requirements make it impractical for us to explore this technique for this dissertation.

5.6.2 Non-static target workloads

The Distiller assumes that the high-level characteristics of the workloads studied do not change drastically over time. This assumed lack of time dependencies allows us to choose any rotate amount when evaluating attribute groups. For example, if analyzed individually, each ten minute segment of a one hour workload ideally has the same attribute-values. Therefore, there should be no difference between rotating a set of parameters ten minutes and rotating that same set of parameters twenty minutes.

In practice, very few workloads have characteristics that do not change at all over time. We must divide those workloads with drastic changes into shorter traces and analyze them separately. The Distiller can handle workloads that exhibit small changes over time; however, we will see in Chapter 7 how these changes contribute to the final synthetic workload's error.

5.6.3 Exploratory workloads provide estimates only

Finally, we emphasize that the techniques we use to evaluate an attribute group provide only an estimate of whether it contains a key attribute. It is possible that two or more attributes in a group have offsetting effects on performance; as a result, the exploratory workloads used when evaluating attribute groups will have low demerit figures in spite of the importance of the offsetting attributes. We believe that the chances of two or more attributes offsetting each other exactly are very low.

5.7 Summary

In this section, we provided the details of our iterative distillation technique. Specifically, we saw how the Distiller iteratively builds a list of key attributes. We also

saw how the Distiller reduces its running time by partitioning the attribute library into attribute groups and exploring only those groups that potentially contain key attributes. The Distiller determines whether an attribute group contains a key attribute by comparing two exploratory workloads: The first preserves almost none of the attributes in the group under test; the second approximately preserves every attribute in that group. The Distiller also uses two exploratory workloads to evaluate individual attributes. The first preserves all attribute-values in the attribute's group, and the second preserves only the value of the attribute under test (for the attribute's group).

CHAPTER 6

EVALUATION OF DISTILLER

In this chapter, we present the results of distilling fifteen target workloads to demonstrate that we have thoroughly tested the Distiller and that it produces representative synthetic workloads when given a sufficient library. In Section 6.1, we distill seven simple *artificial* workloads based upon the Distiller's attribute library. This controlled study stresses different aspects of the Distiller and demonstrates that it works correctly. In Section 6.2 we distilled eight production workloads collected on real production enterprise storage systems. This real-world validation demonstrates that the Distiller and the current attribute library can specify reasonably accurate synthetic versions of real, production workloads.

Section 6.2 also highlights some of the Distiller's limitations. First, the Distiller cannot meet the 10% maximum demerit goal for all workloads. In the worst case, the resulting synthetic workload has a 25% demerit. In addition, the Distiller often can meet the 10% demerit goal for only the log area demerit figure. We also find that the final synthetic workloads have compact representations that are larger than expected: typically 25% to 60% the size of the original workload trace itself. Finally, we see that during each run, the Distiller generates and evaluates between 15 and 125 workloads.

6.1 Artificial workloads

We first verify the correct operation of the Distiller's infrastructure using a set of artificial workloads we generated based on attributes in the Distiller's library. Because the Distiller's library contains all the attributes necessary to synthesize the artificial

workloads, any failure to produce a representative synthetic workload will indicate an error in the Distiller's design or implementation (as opposed to a limitation of the library).

Tables 14 and 6.1 summarize the artificial workloads we used to test the Distiller. We designed these workloads to test our iterative technique's key components. W1 and W2 test the stopping condition. W3 and W4 test choosing and investigating single-parameter attribute groups. W4 demonstrates that the Distiller avoids becoming stuck on local maxima. W5 tests the first-fit attribute search. W6 tests choosing and investigating two-parameter attribute groups; and, W7 demonstrates that attributes need not precisely match the generation to be useful.

Table 15 presents the results of applying the Distiller to each workload using the FC-60 disk array. For these experiments, we configured the Distiller to search for attributes using a first-fit criterion with an RMS demerit threshold of 12%. We fixed the order in which the Distiller evaluated attributes in order to assure that the artificial workloads tested all the Distiller's features.

We now briefly describe each workload and the features of the Distiller tested by that workload. (In the list that follows, the number corresponds to the workload ID shown in Tables 14, 6.1 and 15.)

- **W1** demonstrates that the Distiller will correctly terminate when the intermediate synthetic workload meets the stopping condition. It is a simple workload specified by a uniform distribution of location, a 50/50 distribution of reads and writes, and a constant request size and operation type. The initial attribute list's empirical distribution attributes completely describe this workload; therefore, the Distiller stops before even entering iteration 1.

- **W2** also demonstrates that the Distiller will correctly terminate when the intermediate synthetic workload meets the stopping condition. It is similar to W1,

Table 14: Workload parameters for target artificial workloads.

ID	Location	Operation type	Interarrival time	Request size
W1	uniform [0, 9GB]	constant 20ms	50% reads, 50% writes	constant 8KB
W2	uniform [0, 9GB]	Poisson mean 20ms	33% reads, 66% writes	uniform[1KB, 128KB]
W3	CD(location, location, 1, 4) State 0: U[2MB, 98MB] State 1: U[500MB, 596MB] State 2: U[1GB, 1.096GB] State 3: U[1.5GB, 1.596GB] $\begin{pmatrix} .997 & .001 & .001 & .001 \\ .001 & .997 & .001 & .001 \\ .001 & .001 & .997 & .001 \\ .001 & .001 & .001 & .997 \end{pmatrix}$	CD(op, op, 1,2) State 0: Write State 1: Read $\begin{pmatrix} .97 & .03 \\ .03 & .97 \end{pmatrix}$	CD(iat, iat, 1, 3) State 0: U[.2ms, .9ms] State 1: U[1ms, 5ms] State 3: 250ms $\begin{pmatrix} .80 & .18 & .02 \\ .18 & .80 & .02 \\ .40 & .60 & 0 \end{pmatrix}$	CD(size, size, 1, 4) State 0: 1KB State 1: 16KB State 3: 64KB State 4: 128KB $\begin{pmatrix} .985 & .005 & .005 & .005 \\ .005 & .985 & .005 & .005 \\ .005 & .005 & .985 & .005 \\ .005 & .005 & .005 & .985 \end{pmatrix}$
W4	CD(location, location, 1, 4) State 0: U[2MB, 98MB] State 1: U[500MB, 596MB] State 2: U[1GB, 1.096GB] State 3: U[1.5GB, 1.596GB] $\begin{pmatrix} .997 & .001 & .001 & .001 \\ .001 & .997 & .001 & .001 \\ .001 & .001 & .997 & .001 \\ .001 & .001 & .001 & .997 \end{pmatrix}$	CD(op, op, 1, 2) State 0: Write State 1: Read $\begin{pmatrix} .97 & .03 \\ .03 & .97 \end{pmatrix}$	CD(iat, iat, 1, 3) State 0: U[.2ms, 1ms] State 1: U[5ms, 10ms] State 2: 250ms $\begin{pmatrix} .90 & .08 & .02 \\ .05 & .92 & .03 \\ .20 & .80 & 0 \end{pmatrix}$	CD(size, size, 1, 4) State 0: 1KB State 1: 16KB State 3: 64KB State 4: 128KB $\begin{pmatrix} .985 & .005 & .005 & .005 \\ .005 & .985 & .005 & .005 \\ .005 & .005 & .985 & .005 \\ .005 & .005 & .005 & .985 \end{pmatrix}$

Table 14 (continued)

ID	Location	Operation type	Interarrival time	Request size
W5	CD(location, jump distance, 1, 4) loc range / jump dist. / p(jump) [0GB, 2GB] 1K 98% [2GB, 4GB] 8K 98% [4GB, 6GB] 65K 98% [6GB, 9GB] 128K 98%	CD(op, op, 1, 2) State 0: Write State 1: Read $\begin{pmatrix} .98 & .02 \\ .02 & .98 \end{pmatrix}$	CD(iat, iat, 1, 3) State 0: U[.2ms, .9ms] State 1: U[1ms, 5ms] State 3: 250ms $\begin{pmatrix} .80 & .18 & .02 \\ .18 & .80 & .02 \\ .40 & .60 & 0 \end{pmatrix}$	CD(size, size, 1,4) State 0: 1KB State 1: 16KB State 3: 64KB State 4: 128KB $\begin{pmatrix} .985 & .005 & .005 & .005 \\ .005 & .985 & .005 & .005 \\ .005 & .005 & .985 & .005 \\ .005 & .005 & .005 & .985 \end{pmatrix}$
W6	CD(op, loc, 1, 2) R: 95% [0, 64MB] 5% [65MB, 10GB] W: 5% [0, 64MB] 95% [65MB, 10GB]	CD(op, op, 1, 2) State 0: Write State 1: Read $\begin{pmatrix} .8 & .2 \\ .2 & .8 \end{pmatrix}$	CD(op, iat, 2, 2) W, W: .6ms R, W: 100ms W, R: 25ms R, R: 6ms	CD(op, size, 2, 2) W, W: 128KB R, W: 65KB W, R: 2KB R, R: 16KB
W7	Runs of length Uniform(0,10) [0, 64MB] 90% R, 10% W [65MB,10GB] 10% R, 90% W	CD(op, op, 1, 2) State 0: Write State 1: Read $\begin{pmatrix} .65 & .35 \\ .35 & .65 \end{pmatrix}$	Four interleaved Poisson processes with means: .03ms, .04ms, .05ms, .035ms	Constant(8KB)

Table 15: Results of distilling artificial workloads

ID	Iter.	Attr. group	Attribute added	Result
W1	0		Empirical distributions	3%
W2	0		Empirical distributions	6%
W3	0		Empirical distributions	60%
	1	{loc}	CD(loc, loc, 1, 100)	66%
	2	{op}	CD(op, op, 8, 2)	42%
	3	{size}	CD(size, size, 1, 100)	9%
	4	{iat}	CD(iat, iat, 3, 4)	5%
W4	0		Empirical distributions	15%
	1	{loc}	CD(loc, loc, 2, 10)	22%
	2	{size}	CD(size, size, 1, 100)	9%
W5	0		Empirical distributions	63%
	1	{loc}	CD(loc, jump dist., 1, 100)	23%
	2	{size}	CD(size, size, 1, 100)	13%
	3	{iat}	CD(iat, iat, 1, 100)	11%
W6	0		Empirical distributions	87%
	1	{op,size}	CD(op, size, 1, 2)	54%
	2	{op, loc}	CD(op, loc, 1, 2)	27%
	3	{op, iat}	CD((op,iat), (op, iat), 2, 8)	5%
W7	0		Empirical distributions	78%
	1	{loc}	CD(loc, jump dist., 1, 100)	74%
	2	{op}	CD(op, op, 8, 2)	30%
	3	{op, loc}	CD((op, loc), jump dist, 1, 100)	7%

except interarrival times are distributed exponentially and the request sizes are distributed uniformly between 1KB and 128KB. The initial attribute list's empirical distribution attributes also completely describe this workload; therefore the Distiller stops before even entering iteration 1.

- **W3** tests the Distiller's ability to find single-parameter attributes. Each parameter is generated using a conditional distribution that produce dependencies within the sequence of request parameters, but no inter-parameter dependencies. In other words, the value of each parameter depends on only the values of that parameter for the previous four I/Os.[1] The Distiller explores each single-parameter attribute group and chooses a conditional distribution as a key attribute. After exploring all four single-parameter attribute groups, the intermediate synthetic workload's demerit is below the desired threshold and the Distiller terminates without investigating any two-parameter attribute groups.

- **W4** shows that the Distiller does not fail when the addition of a key attribute produces a temporary degradation in the accuracy of the intermediate synthetic workload. When distilling a workload, the addition of a key attribute does not necessarily produce a more representative intermediate synthetic workload. Two (or more) key attributes may have offsetting effects. For example, a {location} attribute may add sequentiality and speed up the resulting intermediate synthetic workload, while an {interarrival time} attribute added in a later iteration may cause burstiness that slows the intermediate synthetic workload. Consequently, the improvement in the list of key attributes may not become apparent until after the Distiller adds both attributes. The Distiller evaluates all one- and two-parameter attribute groups before reacting to any missing key attributes; therefore, any temporary degradations do not affect its execution.

[1] For demonstration purposes, we set the threshold to 7% so that the Distiller would not terminate after iteration 3.

- **W5** tests the Distiller's ability to explore an attribute group (as opposed to simply choosing the first attribute it tests). During iteration 1, the Distiller tests and rejects three conditional distributions of location and, instead, chooses a conditional distribution of jump distance.

- **W6** tests the Distiller's ability to choose and explore two-parameter attribute groups. In this workload, read and write accesses are concentrated in different areas of the LU's address space and have different request sizes. In addition, successive reads and successive writes have smaller interarrival times than a read followed by a write, or a write followed by a read. The Distiller correctly determines that the are no single-parameter key attributes and explores only two-parameter attribute groups.

- **W7** shows that the Distiller can find a useful set of attributes, even if no attribute corresponds directly to the generation techniques. This workload's run counts vary uniformly from one to ten requests. However, for this experiment, the library did not contained any run count attributes; instead, any run counts must be described using distributions of modified jump distance. Conditional distributions of jump distance can generate only exponential distributions of run counts. The Distiller finds that a modified jump distance attribute sufficiently specifies a representative synthetic workload. Thus, the Distiller shows us that, for this workload, capturing runs is important, but that maintaining the exact run length is not.

Distilling these artificial workloads highlights two of the Distiller's strengths. First, the Distiller properly chooses attributes that produce versions of the artificial synthetic workloads. Second, the Distiller is able to identify key attributes, regardless of the techniques that we used to generate the target artificial workloads.

Table 16: Summary of final synthetic workloads

Workload		External demerit MRT	RMS	LA	Size	Test wklds.	Storage system	Demerit figure
OM	(All)	17%	9%	9%	24%	27 / S	Pantheon	Log area
OM	(2GBLU)	22%	110%	6%	50%	86 / 105	FC-30	RMS
OM	(62GBLU)	11%	24%	16%	60%	103 / 123	FC-30	RMS
OLTP	(All)	23%	145%	24%	55%	51 / 51	Pantheon	Log area
OLTP	(Log)	3%	10%	3%	38%	15 / S	FC-30	RMS
OLTP	(Data)	1%	3%	6%	9%	13 / S	FC-30	RMS
DSS	(1LU)	6%	46%	5%	6%	35 / S	FC-30	Hybrid
DSS	(4LU)	2%	3%	2%	0.2%	13 / S	Pantheon	RMS

6.2 Production workloads

In this section, we evaluate how well our distillation technique (together with the current library of attributes) can specify synthetic workloads that are representative of actual production workloads. While collecting these results, we found that the Distiller was often unable to find a representative synthetic workload when using the RMS demerit figure "internally" to compare exploratory workloads. Therefore, to evaluate the strengths of the Distiller's techniques apart from the weakness of the different demerit figures, we evaluate each workload using the internal demerit figure that produces the most accurate final synthetic workload. Chapter 7 further investigates the effects of using different internal demerit figures.

Based on our experience, we configured the Distiller to perform a first-fit search with a 7.5% threshold. The Distiller evaluates each attribute group's candidate attributes in order of increasing size. We set the stopping condition to be a demerit of 10% for the final synthetic workload. Section 7.2 examines the effects of changing the thresholds.

Table 16 presents a summary of the results. For each workload, it shows

1. the final synthetic workload's MRT, RMS, and log area demerits,

2. the size of the attributes chosen as a percentage of size of the target workload trace,

3. the number of exploratory and intermediate synthetic workloads the Distiller evaluated,

4. the storage system on which we executed the synthetic workloads, and

5. the demerit figure the Distiller used to evaluate attribute groups, individual attributes, and whether the intermediate synthetic workload met the stopping condition.

The first number in the "Test workloads" column is the number of workloads the Distiller evaluated up to the point where it evaluated the final synthetic workload shown in Table 16. For those workloads where the Distiller was unable to meet its 10% demerit goal for the final synthetic workload, the second number shows the total number of workloads the Distiller evaluated before it terminated.

Tables 17 and 18 show the attributes the Distiller chose at each step of its execution. The first section of Tables 17 and 18 lists the name of the workload, the demerit figure the Distiller used internally to compare exploratory workloads, and the threshold. The second section lists the attribute group the Distiller explored during each iteration and the demerit of that attribute group's exploratory workloads. The third section lists the key attribute the Distiller chose and its demerit. The fourth section lists the demerit of the intermediate synthetic workload the Distiller produced at the end of each iteration. Notice that if the Distiller is unable to meet its 10% accuracy goal for the final synthetic workload, the most accurate synthetic workload may come from any iteration. In other words, the last several unsuccessful iterations may reduce the accuracy of the intermediate synthetic workload.

Accuracy: Distiller was able to find synthetic workloads with at most a 10% log area demerit for all but the OpenMail (62GBLU) and OLTP (All) workloads. In both cases, the Distiller's inability to find a highly representative synthetic workload appears to result from a deficiency in the attribute library. For OpenMail (62GBLU),

Table 17: Incremental results of distilling production OpenMail workloads

Workload	Iter.	Attribute group	Error	Attribute added	Error	External demerit MRT	RMS	LA
OM (All)						46%	95%	136%
Log area 7.5%	1	{iat}	20%	β-model ($ws = 5.12$, $il = .01$, $s = 10^{-5}$)	8%	43%	91%	137%
	2	{op. type, loc.}	137%	mJDWS (RW) ($h = 2$, $s = 4$)	6%	**17%**	**9%**	**9%**
OM (2GBLU)						70%	154%	47%
RMS 7.5%	1	{iat}	51%	Cluster ($ws = 5.12$, $bs = .01$, $c = .3$)	10%	44%	132%	23%
	2	{loc.}	41%	CD(loc., loc., 1, 100)	17%	44%	126%	29%
	3	{request size}	29%	TMRC ($h = 1$, $s = 6$)	11%	42%	123%	30%
	4	{op. type}	11%	CD(op. type, op. type, 8, 2)	18%	42%	123%	30%
	5	{op. type, loc.}	32%	mJDWS (RW) ($h = 1$, $s = 100$, *pol*)	8%	**22%**	**110%**	7%
	6	{op. type, req. size}	16%	CD(op. type, req. size, 1, 2)	20%	33%	130%	**6%**
	7	{op. type, iat}	9%	Cluster ($ws = 10.24$, $bs = .0025$, $c = .3$)	10%	25%	123%	14%
OM (65GBLU)						54%	40%	40%
RMS 7.5%	1	{loc.}	60%	mJDWS ($h = 2$, $s = 14$)	15%	71%	50%	63%
	2	{iat}	37%	Cluster ($ws = 10.24$, $bs = .0025$, $c = .3$)	4%	58%	45%	51%
	3	{req. size}	13%	CD(request size, request size, 2, 10)	5%	45%	25%	40%
	4	{op. type}	13%	Operation type STC	12%	46%	26%	41%
	5	{op. type, req. size}	113%	Joint dist. of op. type and reqt size	116%	44%	**24%**	40%
	6	{op. type, loc.}	32%	RCWS (RW) ($h = 2$, $s = 100$)	8%	**11%**	28%	**16%**

Table 18: Incremental results of distilling production OLTP and DSS workloads

						External demerit		
Workload	**Iter.**	**Attribute group**	**Error**	**Attribute added**	**Error**	**MRT**	**RMS**	**LA**
OLTP (All)						40%	184%	200%
Log area 7.5%	1	{op type}	35%	Read percentage	6%	41%	195%	154%
	2	{op. type, loc.}	174%	CD((op. type, loc.), loc., 2, 100)	30%	**23%**	**145%**	32%
	3	{loc., iat}	20%	CD(loc., iat, 3, 2)	21%	24%	155%	**24%**
OLTP (log)						7%	34%	10%
RMS 7.5%	1	{loc.}	16%	mJDWS ($h = 1$, $s = 1$)	7%	**3%**	**10%**	**3%**
OLTP (data)						4%	10%	21%
RMS 7.5%	1	{iat}	19%	CD(iat, iat, 2, 10)	8%	**1%**	**3%**	**6%**
DSS (1LU)						0%	46%	30%
Hybrid 7.5%	1	{loc.}	67%	Run count within state ($h = 1$, $s = 25$)	5%	19%	55%	15%
	2	{iat}	31%	Cluster($ws = 5.12$, $bs = .01$, $c = .1$)	1%	**6%**	**47%**	**5%**
DSS (4LU)						14%	24%	17%
RMS 7.5%	1	{loc.}	17%	IR ($mg = 2048$)	1%	**2%**	**3%**	**2%**

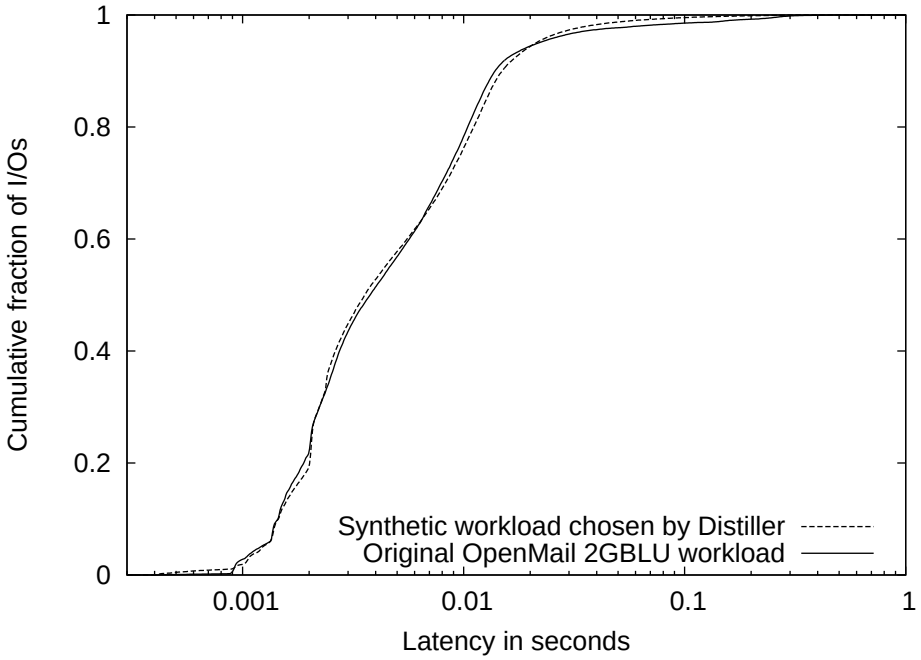

Figure 23: Final synthetic workload for OpenMail 2GBLU

the most accurate {operation type, request size} attribute has an RMS demerit of 116%. For OLTP (All) the most accurate {operation type, location} and {operation type, interarrival time} attributes have demerits of 30% and 21% respectively.

Several synthetic workloads had RMS demerits over 10%. The large RMS demerit figure for OpenMail (62GBLU) and OLTP (All) reflect the limitations of the attribute library. For OpenMail (2GBLU), the workload's heavy tail appears to be the primary cause of the large error values. Figure 23 as well as the log area demerit of only 1% indicate that the synthetic workload is not as inaccurate as the RMS demerit of 14% suggests. Finally, we distill OLTP (All) and DSS (1LU) using the log area and hybrid demerit figures respectively. As a result, the Distiller chooses attributes that minimize these demerit figures. These attributes may not produce the lowest RMS demerit. Chapter 7 explores how the choice of demerit figure and other configuration decisions affect the quality of the Distiller's answers.

Replay and randomness errors: To help put the error of each final synthetic workload in perspective, Tables 19 and 20 present the replay and randomness errors of

Table 19: Replay errors of production workloads

Workload		MRT			RMS			Log area		
		Mean	70%	Max	Mean	70%	Max	Mean	70%	Max
OM	(2GBLU)	2%	2%	7%	14%	11%	44%	1%	1%	1%
OM	(62GBLU)	3%	5%	4%	6%	9%	15%	1%	1%	1%
OLTP	(Log)	0%	2%	1%	3%	4%	7%	0%	0%	0%
OLTP	(Data)	1%	1%	1%	1%	1%	2%	1%	1%	2%
DSS	(1LU)	1%	1%	3%	3%	5%	5%	1%	1%	3%

Table 20: Randomness errors of production workloads

Workload		MRT			RMS			Log area		
		Mean	70%	Max	Mean	70%	Max	Mean	70%	Max
OM	(All)	3%	4%	6%	1%	2%	2%	1%	2%	3%
OM	(2GBLU)	2%	3%	5%	8%	9%	17%	1%	1%	1%
OM	(62GBLU)	2%	2%	3%	5%	6%	8%	2%	2%	4%
OLTP	(All)	4%	5%	8%	49%	67%	92%	6%	8%	9%
OLTP	(Log)	2%	3%	4%	6%	9%	13%	0%	1%	1%
OLTP	(Data)	1%	1%	2%	3%	4%	5%	2%	1%	2%
DSS	(1LU)	11%	15%	22%	31%	40%	58%	8%	11%	15%

the workloads studied. (Note: Pantheon is deterministic; therefore, workloads studied using Pantheon have no replay error.) For those workloads studied using FC-30, the replay error is the minimum possible error that one can obtain without multiple trials for each workload evaluated. (It is not possible to determine a synthetic workload has less error than the inherent measurement error.) For all workloads, the randomness error reflects the amount of error introduced by varying the random seed used when generating the final synthetic workload.

Only OpenMail (2GBLU) has a large replay error. The otherwise low replay errors indicate that (1) replay error is not the primary cause of the high demerits in Table 16, and (2) it should be possible, given a sufficient attribute library, to obtain a synthetic workload with a demerit of at most 10%.

Only the two DSS workloads have large randomness errors for all three demerit figures. We suspect that this large error has two causes: First, the Distiller specifies synthetic workloads using two and one attributes respectively. These terse descriptions leave more room for variation than a more detailed compact representation.

Second, the DSS workload produces the heaviest overall load on the disk array. The DSS workloads are read-only and access very few locations more than once. As a result, almost every request requires a physical disk access. The response times for operations that involve physical movement can vary by milliseconds depending on how far the disk components must move. Therefore, this workload is likely to have a larger randomness and replay errors than other workloads.

Although using multiple trials may reduce a synthetic workload's randomness error, we suspect that the better solution is to implement an attribute that allows for less variation in the synthetic workload. However, such a solution must balance the benefits of a reduced randomness error with the cost of a larger compact representation. We leave the study of the randomness error / attribute size tradeoff for future work.

Compact representation size: We find that, in practice, the final synthetic workloads have compact representations that are 25% to 60% the size of the target workload trace. Our precise specification of the workload's footprint cause these large compact representations. For example, the distributions of read and write location for OLTP (All) represent 99% of the final synthetic workload's compact representation. In Chapter 7.3, we study the tradeoff between the precision (and size) of a synthetic workload's compact representation and its accuracy.

Running time: Generating and evaluating exploratory workloads dominates the Distiller's running time. Therefore, we specify the running time by counting these workloads. In general, we find that the Distiller has a much shorter running time when it can meet its accuracy goal. This reduced running time is expected because the Distiller stops exploring attribute groups when it meets it's goal.

More importantly, for the Distiller not to meet its accuracy goal, the library must lack key attributes for one or more attribute groups. As a result, the Distiller will evaluate every attribute in the group while searching for the key attribute. It is these

long searches through the attribute library that cause the Distiller's long running times. For example, no {interarrival time} attribute meets the 7.5% RMS threshold for OpenMail (2GBLU). Consequently, the Distiller evaluates all 26 {interarrival time} attributes. In contrast, most workloads for which the library contains a sufficient {interarrival time} attribute require evaluations of at most five {interarrival time} attributes.

6.3 Summary

In this chapter, we demonstrated that we can use our technique of iterative distillation to automatically find the attributes that specify representative synthetic workloads. Section 6.1 verified the correct operation of the Distiller's infrastructure by successfully choosing attributes for seven artificial target workloads. In each case, the resulting final synthetic workload met the target of a RMS demerit of at most 12%. Section 6.2 demonstrated that we can use the Distiller and its library to specify synthetic workloads representative of real production workloads. In particular, we saw that the Distiller was able to specify a final synthetic workload with a log area demerit of 10%, in most cases, and 25% in the worst case.

We also saw several of the Distiller's limitations. In particular, (1) the Distiller was not able to meet its 10% accuracy for all workloads, nor was it able to meet its 10% accuracy goal for several different demerit figures (most notably RMS); (2) the Distiller tends to choose attributes with large representations; and (3) the Distiller tends to have a long running time when the library is insufficient for the workload under test.

CHAPTER 7

OPERATIONAL SENSITIVITY

Before running the Distiller, the user must make two decisions: (1) which demerit figure (mean response time, RMS, log area, hybrid) to use when comparing exploratory workloads, and (2) what threshold to set for choosing attributes and exploring attribute groups. After running the Distiller, the user can explicitly trade some of the final synthetic workload's accuracy for a reduction in the size of its compact representation. This chapter examines the consequences of these decisions.

In Section 7.1, we see that the hybrid demerit figure leads the Distiller to near-optimal final synthetic workloads in most of our test cases, but that the log area's limitations affect the distillation process less dramatically than the other demerit figures. In Section 7.2, we see that raising the internal demerit threshold reduces the quality of the final synthetic workload, but also reduces the Distiller's running time and the size of the final synthetic workload's compact representation. Section 7.3 shows that we can run the Distiller in a "post-process mode" and reduce the final synthetic workload's compact representation by over 70% while reducing the accuracy by less than 20%.

7.1 Demerit figure

This section discusses how the limitations of the Distiller interact with the limitations of the demerit figures and affect the distillation process. In Section 7.1.1, we see that when using the MRT or hybrid demerit figures, randomness error and the target workload's non-static workload characteristics can lead the Distiller to different answers based on its initial random seed. In Section 7.1.2, we see that the RMS demerit

Table 21: Effects of modifying the demerit figure

Workload		Internal demerit figure	External demerit MRT	RMS	LA	Hybrid	Size	Test wklds.
OM	(All)	MRT	**22%**	33%	22%	22%	24%	35 / 37
		RMS	20%	**26%**	15%	20%	25%	63 / 90
		Log Area	17%	9%	**9%**	17%	24%	27 / S
		Hybrid	19%	26%	10%	**19%**	38%	54 / 67
OLTP	(All)	MRT	**31%**	136%	102%	102%	67%	37 / 37
		RMS	31%	**185%**	59%	59%	26%	18 / 96
		Log Area	23%	145%	**24%**	24%	55%	51 / 51
		Hybrid	9%	68%	22%	**30%**	67%	45 / 61
DSS	(4LU)	MRT	**5%**	3%	3%	5%	0.2%	12 / S
		RMS	2%	**3%**	2%	2%	0.2%	12 / S
		Log Area	5%	3%	**3%**	5%	0.2%	12 / S
		Hybrid	5%	3%	3%	**5%**	0.2%	12 / S

figure's emphasis on tail behavior causes it to ignore differences between attributes that have a noticeable effect on the accuracy of the final synthetic workload. From these observations, we conclude that, although it has its own limitations, log area is currently the best of the four demerit figures tested.

Table 21 shows the effects of varying the Distiller's internal demerit figure. To collect these results, we ran the Distiller several times for each workload. Each run differed only in the demerit figure the Distiller used internally to select attribute groups and attributes. In each case, the threshold was 7.5%.

For each external demerit figure, we present the lowest demerit over all of the intermediate synthetic workloads (i.e., the synthetic workloads produced at the end of each iteration). Consequently, the external demerits in Table 21 are not necessarily all from the same intermediate synthetic workload. For example, when using RMS internally to distill OpenMail (All), the intermediate synthetic workload produced after iteration 5 has the lowest MRT demerit (20%), whereas the intermediate synthetic workload produced after iteration 4 has the lowest log area demerit (15%). Tables 22 through 24 show the attributes chosen at each iteration and the external demerits of each resulting intermediate synthetic workload.

Table 22: Incremental results of distilling OM using different demerit figures

Workload	Iter.	Attribute group	Error	Attribute added	Error	External demerit MRT	RMS	LA
OM (All)						49%	96%	140%
RMS 7.5%	1	{loc.}	20%	mJDWS($s = 1$, $s = 100$)	8%	54%	100%	144%
	2	{iat}	17%	β-model ($ws = 5.12$, $il = .01$, $s = 10^{-2}$)	3%	44%	97%	155%
	3	{op. type, loc.}	72%	mJDWS (RW) ($h = 1$, $s = 100$, pol)	14%	24%	**26%**	30%
	4	{op. type, iat}	11%	Cluster($ws = 10.24$, $bs = .01$, $c = .3$)	9%	25%	26%	**15%**
	5	{loc., req. size}	30%	CD(loc., req. size, 1, 100)	23%	**20%**	26%	17%
OM (All)						46%	95%	136%
Log area 7.5%	1	{iat}	20%	β-model ($ws = 5.12$, $il = .01$, $s = 10^{-5}$)	8%	43%	91%	137%
	2	{op. type, loc.}	137%	mJDWS (RW) ($h = 2$, $s = 4$)	6%	**17%**	**9%**	**9%**
OM (All)						48%	96%	137%
Hybrid 7.5%	1	{iat}	20%	Cluster ($ws = 5.12$, $bs = .01$, $c = .3$)	1%	50%	92%	120%
	2	{loc.}	9%	Dist. clustering (ws=16MB, bs=8192B, $c = .2$)	7%	47%	92%	120%
	3	{op. type, loc.}	136%	mJDWS ($h = 1$, $s = 2$, pol)	5%	**19%**	**26%**	**10%**
	4	{op. type, req. size}	12%	CD(op. type, req. size, 1, 2)	3%	39%	37%	10%
	5	{op. type, iat}	9%	Cluster ($ws = 2.56$, $bs = .01$, $c = .3$)	7%	27%	32%	10%
OM (All)						47%	95%	136%
MRT 7.5%	1	{loc.}	22%	Run count within state ($h = 1$, $s = 25$)	3%	27%	88%	126%
	2	{iat}	10%	β-model ($ws = 5.12$, $il = .01$, $s = 10^{-2}$)	1%	24%	86%	141%
	3	{op. type, loc.}	78%	CD(op. type, loc., 1, 2)	4%	27%	34%	27%
	4	{loc., req. size}	12%	CD(op. type, req. size, 1, 2)	4%	**22%**	34%	30%
	5	{op. type, iat}	9%	CD(op. type, iat, 1, 1)	5%	22%	**33%**	**22%**

Table 23: Incremental results of distilling OLTP using different demerit figures

Workload	Iter.	Attribute group	Error	Attribute added	Error	External demerit MRT	RMS	LA
OLTP (All)						41%	194%	157%
RMS 7.5%	1	{iat}	116%	β-model ($ws = 5.12$, $il = .01$, $s = 10^{-5}$)	4%	40%	**185%**	152%
	2	{op. type}	43%	Read percentage	42%	40%	185%	173%
	3	{request size}	18%	CD (req. size, req. size, 4, 4)	7%	40%	192%	174%
	4	{loc.}	12%	CD(loc., loc., 1, 1)	15%	40%	192%	174%
	5	{op. type, loc.}	95%	mJDWS (RW) ($h = 1, s = 100$)	23%	61%	345%	61%
	6	{op. type, req. size}	61%	CD(op. type, req. size, 2, 50)	32%	**31%**	200%	70%
	7	{op. type, iat}	57%	CD(op. type, iat, 2, 1)	31%	40%	240%	**59%**
OLTP (All)						40%	184%	200%
Log area 7.5%	1	{op type}	35%	Read percentage	6%	41%	195%	154%
	2	{op. type, loc.}	174%	CD((op. type, loc.), loc., 2, 100)	30%	**23%**	**145%**	32%
	3	{loc., iat}	20%	CD(loc., iat, 3, 2)	21%	24%	155%	**24%**
OLTP (All)						39%	182%	197%
Hybrid 7.5%	1	{op. type}	11%	Read percentage	11%	48%	244%	165%
	2	{iat}	10%	β-model ($ws = 5.12$, $il = .01$, $s = 10^{-5}$)	1%	40%	187%	176%
	3	{op. type, loc.}	174%	CD(op. type, loc., 1, 2)	27%	30%	209%	**22%**
	4	{op. type, request size}	8%	Joint distribution of op. type and req. size	5%	**9%**	97%	47%
	5	{loc., iat}	20%	CD(loc., iat, 2,10)	20%	15%	**68%**	46%
OLTP (All)						41%	193%	163%
MRT 7.5%	1	{iat}	14%	β-model ($ws = 5.12$, $il = .01$, $s = 10^{-5}$)	0%	46%	232%	154%
	2	{op. type}	12%	Read percentage	9%	40%	185%	172%
	3	{op. type, request size}	8%	CD(op. type, request size, $h = 1$, $s = 2$)	6%	**31%**	**136%**	145%
	4	{op. type, loc.}	8%	mJDWS (RW) ($h = 2, s = 4$)	5%	49%	268%	**102%**

Table 24: Incremental results of distilling DSS using different demerit figures

Workload		Group	Error	Attribute added	Error	MRT	RMS	LA
DSS (4LU)						14%	24%	17%
RMS 7.5%	1	{loc.}	17%	IR ($mg = 2048$)	1%	**2%**	**3%**	**2%**
DSS (4LU)						14%	24%	18%
LA 7.5%	1	{loc.}	14%	IR ($mg = 2048$)	3%	**5%**	**3%**	**3%**
DSS (4LU)						14%	24%	18%
Hybrid 7.5%	1	{loc.}	14%	IR ($mg = 2048$)	3%	**5%**	**3%**	**3%**
DSS (4LU)						14%	24%	18%
MRT 7.5%	1	{loc.}	10%	IR ($mg = 2048$)	1%	**5%**	**3%**	**3%**

The "Size" column in Table 21 presents the the size of the final synthetic workload for the external demerit figure that matches the internal demerit figure under test. For example, when using log area internally, the "Size" column refers to the intermediate synthetic workload with the lowest log area demerit. (Although each external demerit figure may correspond to a separate final synthetic workload and, hence, have a separate size, we find that presenting all four sizes clutters the table.)

The "Test workloads" column includes two figures. The first shows the number of exploratory workloads the Distiller generated and evaluated up through the iteration during which it generated the final synthetic workload (for which the internal and external demerit figures match). When the Distiller cannot meets its accuracy goal, it chooses the intermediate synthetic workload with the lowest demerit to be the final synthetic workload. In this case, the second figure is the total number of workloads the Distiller generated and evaluated during its execution (including both exploratory and intermediate synthetic workloads). A second value of "S" indicates that the Distiller succeeded in meeting its 10% accuracy goal.

7.1.1 Distiller design limitations

In this section, we see that the Distiller's three main limitations (discussed in Section 5.6) significantly affect its execution when using the MRT or hybrid demerit figures internally. In particular, we see how the subtractive workloads' randomness

Table 25: Effects of subtractive workload's randomness error on {location} evaluation

	Internal demerit figure					Internal demerit figure			
Test	MRT	RMS	LA	Hybrid	Test	MRT	RMS	LA	Hybrid
1	7.6%	19%	2.8%	7.6%	9	10.0%	20.2%	3.1%	10.0%
2	12.7%	19.7%	3.0%	12.7%	10	12.1%	19.3%	2.8%	12.1%
3	17.2%	18.9%	2.7%	17.2%	11	18.8%	19.0%	2.9%	18.8%
4	**7.5%**	20.2%	2.9%	**7.5%**	12	**6.6 %**	19.4%	2.8%	**6.6%**
5	21.8%	19.7%	3.1%	21.8%	13	19.8%	19.1%	3.0%	19.8%
6	12.9%	19.8%	3.1%	12.9%	14	9.2%	19.8%	2.9%	9.2%
7	18.4%	19.5%	3.0%	18.4%	15	21.3%	18.4%	2.8%	21.3%
8	14.1%	20.3%	3.1%	14.1%					

errors affect the Distiller's ability to estimate whether a given attribute group contains a key attribute, and how the target workloads' non-static behaviors affect which attributes the Distiller chooses as key attributes. In some cases, these effects cause the Distiller to produce final synthetic workloads with very different accuracy depending on its initial random seed. In contrast, the Distiller's limitations have much smaller effects on its execution when using the RMS or log area demerit figures internally.

Randomness errors: Table 25 shows that, when using the MRT or hybrid demerit figures internally, the subtractive {location} workload's randomness error affects whether the Distiller explores the {location} attribute group. Specifically, Table 25 shows the results of evaluating the {location} attribute group fifteen times using different random seeds when generating the subtractive exploratory workload. The MRT and hybrid demerits vary from 6.6% to 21.3%. In contrast, the RMS demerit varies from 18.4% to 20.3%; and the log area demerit varies from 2.7% to 3.1%. The MRT and hybrid differences will greatly affect the Distiller's execution because it will explore the {location} attribute group during only 87% of its executions (given a 7.5% threshold). In contrast, the Distiller will always explore the {location} attribute group when using the RMS demerit figure and never explore {location} when using log area.

One technique for reducing the effects of randomness error is to generate and

Table 26: Difference between CDFs for rotated {location} workloads

		RMS			Log Area			MRT		
Wkld.	PF	Min	Max	Stdev.	Min	Max	Stdev.	Min	Max	Stdev.
OM	0KB	91%	93%	.005	24%	27%	.007	28%	54%	.06
OLTP	0KB	149%	289%	.25	108%	147%	.09	30%	53%	.45
DSS	0KB	0%	2%	.003	0%	1%	.001	0%	1%	.03
OM	128KB	87%	89%	.005	110%	113%	.006	29%	53%	.06
OLTP	128KB	114%	393%	0.33	36%	124%	.11	24%	88%	.68
DSS	128KB	4.2%	877%	1.37	2%	41%	.07	0%	93%	4.85

execute several versions of each exploratory workload (using a different random seed for each version), then compute the "average" CDF using the technique discussed in Section 3.4.2. The main challenge in repeating evaluations for statistical confidence is that the Distiller requires considerable CPU and storage resources to produce and evaluate a synthetic workload. For larger test workloads, repeating each evaluation is not practical.

Rotate amount and non-static behaviors: When evaluating attributes, the Distiller generates an exploratory workload by rotating one or more parameters. The Distiller's design assumes that the amount by which the Distiller rotates parameters does not affect the result of attribute group evaluations; however, this assumption does not hold for most workloads. Fortunately, the rotate amount causes only minor differences in the RMS and log area demerits. The differences measured by MRT are much larger.

Table 26 shows the minimum and maximum demerits that result from comparing a rotated {location} workload with the target workload. Specifically, we generated 99 workloads by rotating location in 1% increments. For OpenMail and DSS, the differing rotation amounts affect the RMS and log area demerits by at most 3%. For OLTP, the differences are much larger: 140% for RMS and 39% for log area. In contrast, the difference between the largest and smallest MRT (and hybrid) demerits is between 20% and 25% for OpenMail and OLTP.

We also repeated this test using a slightly modified disk array. In particular, we

Table 27: Attribute chosen given different rotate amounts

Rotate amount	Attribute chosen RMS	MRT	LA	Hybrid	Attr. ID	Description
-25%	A3	A1	A4	A1	A1	CD(op. type, location, 1, 2)
-15%	A3	A5	A4	A2	A2	mJDWS (RW) ($h = 1$, $s = 1$)
-10%	A3	A5	A4	A2	A3	mJDWS (RW) ($h = 1$, $s = 100$)
10%	A3	A5	A2	A2	A4	mJDWS (RW) ($h = 2$, $s = 4$)
15%	A3	A1	A4	A2	A5	mJDWS (RW) ($h = 2$, $s = 10$, *pol*)
25%	A3	A1	A4	A3		

set prefetching to 128KB. (Section 8.1 discusses prefetching in detail.) We see that this modification does not change the effects of rotate amount much for OpenMail or OLTP, but drastically changes the effects of rotate amount for DSS. In fact, the differences in RMS and MRT demerits are so large that we currently are unable to use the Distiller to study the DSS workload when prefetching is set to 128KB.

We now examine how the differing demerits affect the Distiller's execution. When evaluating {operation type, location} attributes, the Distiller rotates the request size and interarrival time attributes backward 25% (thereby breaking any correlations with request size and interarrival time without changing the workload's locality). To test the sensitivity to the rotate amount, we generated workloads by rotating the request sizes and interarrival times -25%, -15%, -10%, 10%, 15%, and 25%. We then determined which attribute the Distiller would choose when using each rotate amount. Table 27 shows the results for OpenMail (All). With one exception, the rotate amount does not affect the choice of attribute when using RMS or log area. When using log area, the Distiller chooses mJDWS (RW) ($h = 2$, $s = 4$) except when the rotate amount is 10%. In this case, it chooses mJDWS (RW) ($h = 1$, $s = 1$). Thus, it chooses the same attribute with a different configuration. In contrast, when using the MRT demerit figure, the Distiller chose one of two completely different attributes ($A1$ or $A5$) depending on the rotate amount.

To determine the effect of the rotate amount on the Distiller's final answer, we

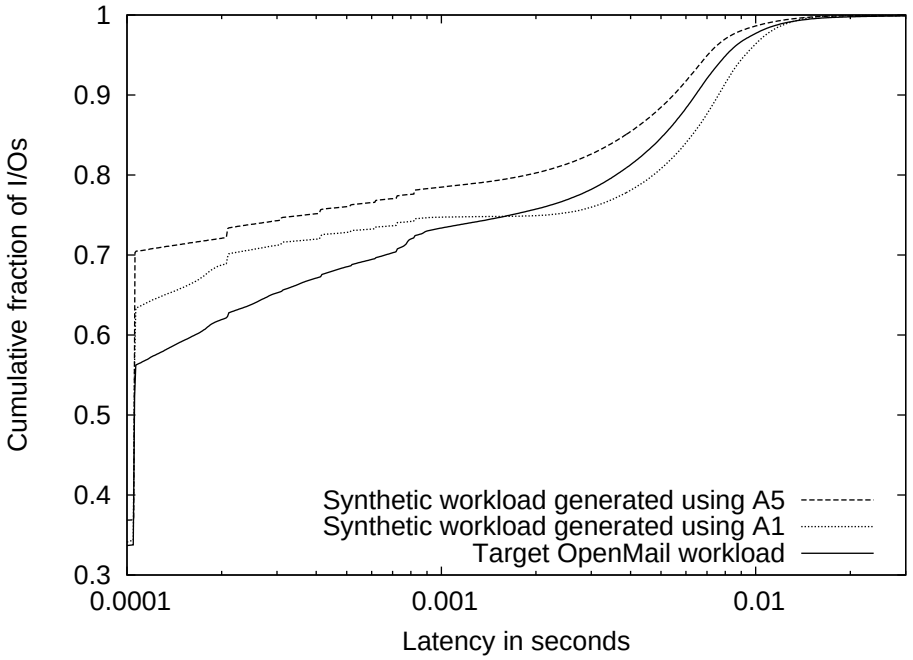

Figure 24: Effects of choosing A5 instead of A1 when using MRT

Table 28: Effects of choosing A5 instead of A1 when using MRT

Attribute	Attribute configuration	MRT	RMS	LA	Hybrid
A1	CD(op. type, location, 1, 2)	22%	33%	22%	22%
A5	mJDWS (RW) ($h = 1$, $s = 100$, pol)	5%	39%	66%	66%

Table 29: Comparison of test workloads illustrating differences between RMS and log area.

Workloads		Demerit figure MRT	RMS	LA	Hybrid
Test 1	Test 2	1.1%	2%	36%	36%
Test 1	Test 3	5.4%	60%	1%	5.4%
Test 2	Test 3	4.3%	60%	33%	33%

generated a final synthetic workload using attribute $A5$ and compared it to the final synthetic workload that the Distiller initially generated using attribute $A1$. Figure 24 and Table 28 show that the demerits for the workloads generated using $A1$ and $A5$ are quite different. Hence, we see that the time-dependent behavior in OpenMail is large enough to seriously affect the result of running the Distiller with the MRT demerit figure.

7.1.2 Demerit figure limitations

This section shows how the log area demerit figure can overcome the limitations of the Distiller's library and the limitations of the RMS demerit figure. Specifically, we show how the log area demerit figure's equal emphasis of errors at all times scales and its tendency to choose attributes whose errors offset the errors in the intermediate synthetic workload allow it to produce final synthetic workloads with lower RMS demerits than the final synthetic workloads produced using the RMS demerit figure internally.

RMS and log area differences: To understand how RMS and log area affect distillation process, we must first understand their main differences. The log area demerit figure measures relative differences whereas the RMS demerit figure measures absolute differences. As a result, RMS emphasizes differences that occur in the tails of response time distributions. Figures 25 and 26 show three hypothetical CDFs of response time. The spikes at $x = .0001s$ represent cache hits. The slope from $x = .0001s$ to $x = .02s$ represent I/Os that miss in the cache and suffer seek time.

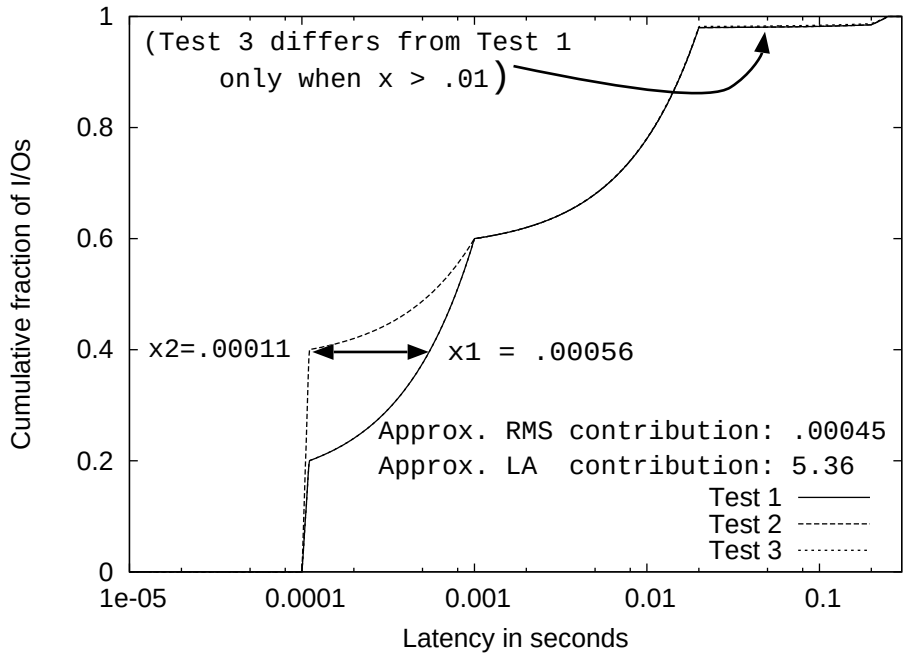

Figure 25: Differences between CDFs affect RMS and log area demerit figures differently

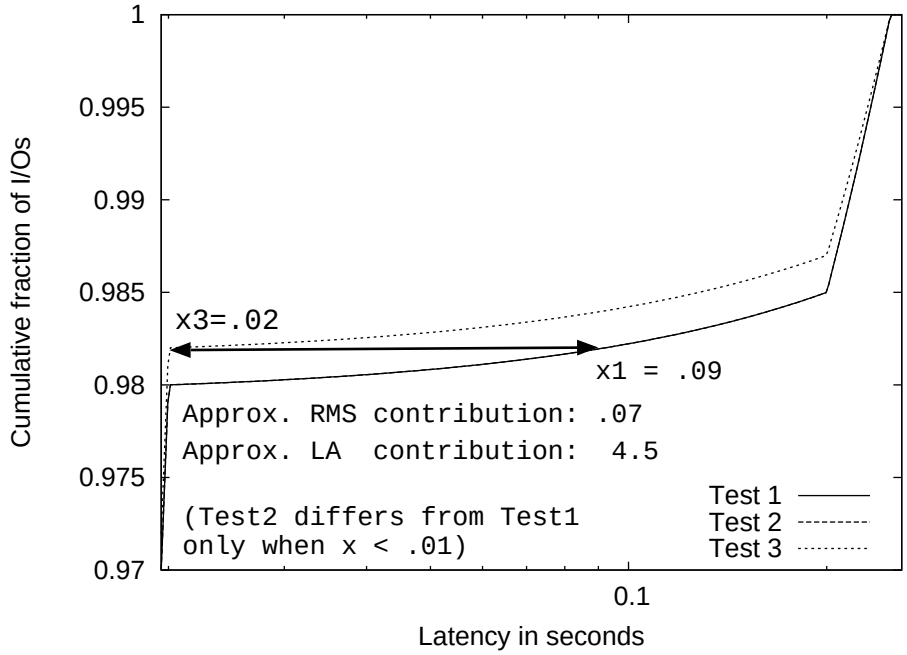

Figure 26: Differences between CDFs affect RMS and log area demerit figures differently (focus on tail)

Table 30: Comparison of {interarrival time} attributes chosen using log area and RMS demerit figures

Internal dmt. fig.	Attr.	Configuration	Internal demerit of attribute MRT	RMS	LA
RMS	$A6$	β-model ($ws = 5.12$, $bs = .01$, $s = 10^{-5}$)	0%	3%	19%
LA	$A7$	β-model ($ws = 5.12$, $bs = .01$, $s = 10^{-2}$)	2%	4%	7%

The time required for the disk heads to move to the physical locations of the data dominates the latency of these I/Os. Finally, the slope from $x = .2s$ to $x = .25s$ represents I/Os that are delayed in a queue. The queuing delay dominates the latency of these I/Os. Table 29 provides the demerit figures for these workloads.

Workloads Test 1 and Test 2 are identical, except that workload Test 2 has a cache hit rate of 40% while Test 1 has a cache hit rate of only 20%. We can see in Figure 25 that the absolute horizontal difference when $y = 0.4$ is approximately $.00056s - .00011s = .00045s$, or approximately 5.5% of Test 1's $.0082s$ mean response time. The resulting RMS demerit for Test 2 is 2%. In contrast, the relative difference is $.00056s/.00011s = 510\%$ contributing to a log area demerit of 36%. Thus, we see how the RMS demerit figure is affected little by large differences in cache behavior.

Workloads Test 1 and Test 3 are identical, except workload Test 3 has 0.2% more seeks and 0.2% fewer queued I/Os than workload Test 1. Figure 26 shows that the absolute horizontal difference when $y = .982$ is approximately $.09s - .02s = .07s$, or 830% of Test 1's $.0082s$ mean response time. The resulting RMS demerit for Test 3 is 60%. In contrast, the relative difference is $.09s/.02s = 450\%$, contributing to a log area demerit of only 1%. Thus, we see how the log area demerit figure is affected little by differences in tail behavior.

We now provide two examples of how the differences between RMS and log area affect the Distiller's execution and how using log area internally can produce a final synthetic workload with a lower RMS demerit than running the Distiller using the RMS demerit figure internally.

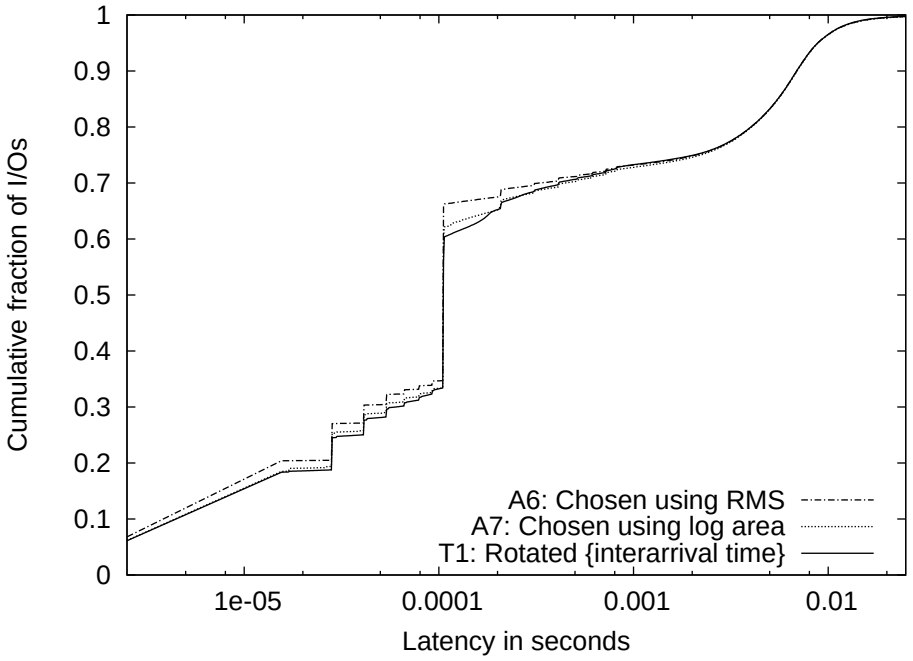

Figure 27: Comparison of {interarrival time} attributes chosen using log area and RMS demerit figures

"Internal" vs. "external" error: Log area's equal emphasis of differences at all time scales allows it to distinguish between two attributes that have similar RMS values. Differences that do not affect the "internal" RMS demerit (when comparing attributes to the rotated workload for the attribute group under test) can occasionally later affect the "external" RMS demerit (when evaluating an intermediate or final synthetic workload). In these cases, using the log area demerit figure internally leads to a final synthetic workload with a lower RMS demerit that the final synthetic workload produced using RMS internally.

We can see one example of this phenomenon by comparing the {interarrival time} attributes the Distiller chose for OpenMail when using RMS and log area internally. In particular, Figure 27 and Table 30 show the behavior of two synthetic arrival patterns. Both are generated using Wang's β-model [56]; however, when using the RMS demerit figure, the Distiller specifies a sensitivity of $10^{-5}s$ (henceforth called $A6$), and when using log area, it specifies a sensitivity of $10^{-2}s$ (henceforth called $A7$).

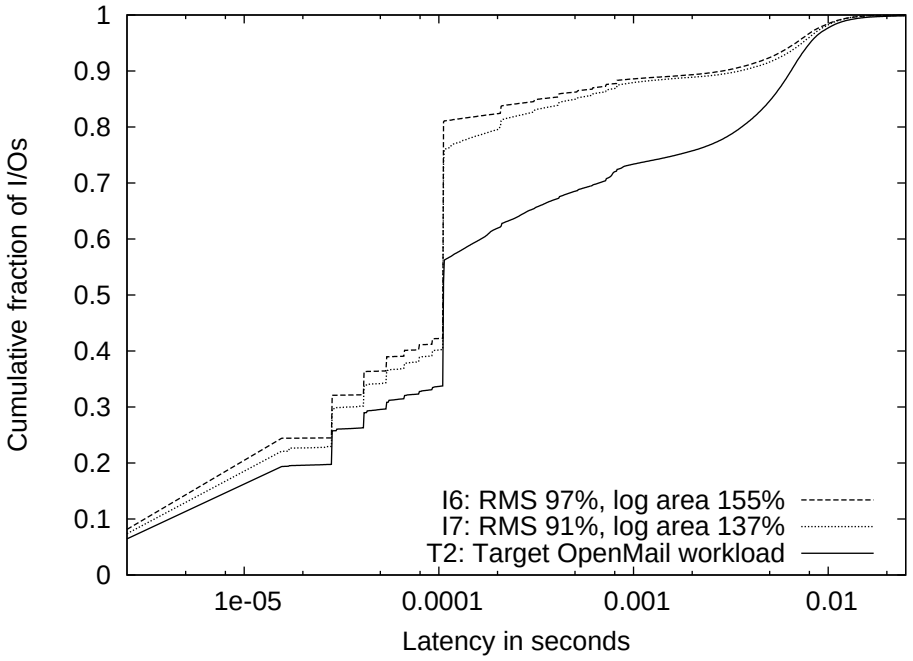

Figure 28: Comparison of intermediate synthetic workloads using log area and RMS

(We discuss the parameters for the β-model in Section 3.3.) The Distiller compares these workloads to the rotated {interarrival time} workload (henceforth called $T1$).

Attributes $A6$ and $A7$ have similar RMS demerits, but their log area demerits differ by a factor of about 2.5. Because $A6$ and $A7$ differ from $T1$ only when $x < .001s$, the RMS demerits for each workload are small. The visually noticeable difference at $y = 0.66$ has little effect on $A6$'s RMS demerit. In contrast, the relative difference at $y = .66$ is 100%, contributing to a 19% log area demerit for $A6$.

Even though $A6$ and $A7$ have similar RMS demerits, they lead to intermediate synthetic workloads with noticeably different RMS demerits. After choosing a key {interarrival time} attribute, the Distiller adds the chosen attribute to the intermediate attribute list and generates intermediate synthetic workloads (call them $I6$ and $I7$). We can see in Figure 28 how attributes $A6$ and $A7$ influence the performance of $I6$ and $I7$ (especially when $.7 < y < .85$). The intermediate attribute lists are not yet complete (the Distiller has not yet investigated all attribute groups); therefore, $I6$ and $I7$ are not very accurate (i.e., not similar to $T2$, the target OpenMail workload).

The large absolute differences between $I6$ and $T2$ and between $I7$ and $T2$ noticeably influence the RMS demerit — especially when $.7 < y < .85$. Furthermore, because the RMS demerit figure squares horizontal distances, the additional horizontal difference between $I6$ and $T2$ (as compared to $I7$) causes the difference in RMS demerits for $I6$ and $I7$ to be six times larger than the difference in RMS demerits for $A6$ and $A7$. Thus, we see how differences between attributes can have a small effect on RMS "internally" (because the exploratory workloads compared are similar), but then have a much larger effect "externally" (because the intermediate synthetic workload and the target workload are not similar).

The difference between the internal and external RMS demerits explains why the Distiller produces synthetic workloads with lower RMS demerits when using the log area demerit figure internally than when using the RMS demerit figure internally. In this case, the RMS demerit figure failed to reflect the differences between $A6$ and $A7$ internally, even though those differences had a significant effect on the external RMS demerit. In contrast, the log area demerit figure did reflect those differences, thereby reducing both the log area and RMS external demerits.

Offsetting errors: If the Distiller's library of attributes does not contain an accurate attribute for some attribute group, the Distiller adds the most accurate available attribute to the intermediate attribute list. These "most accurate" attributes, when chosen using the log area demerit figure, tend to have errors that offset the errors in the intermediate synthetic workload (caused by missing patterns described by attributes from attribute groups not yet explored). Consequently, when the Distiller adds them to the intermediate attribute list, the resulting synthetic workload is fairly accurate. Given the many ways in which CDFs of response time can differ, we initially expected this situation to be rare; however, it occurred when distilling both OpenMail and OLTP. We now present two examples of offsetting errors.

Consider the choice of attribute for the {operation type, location} attribute group

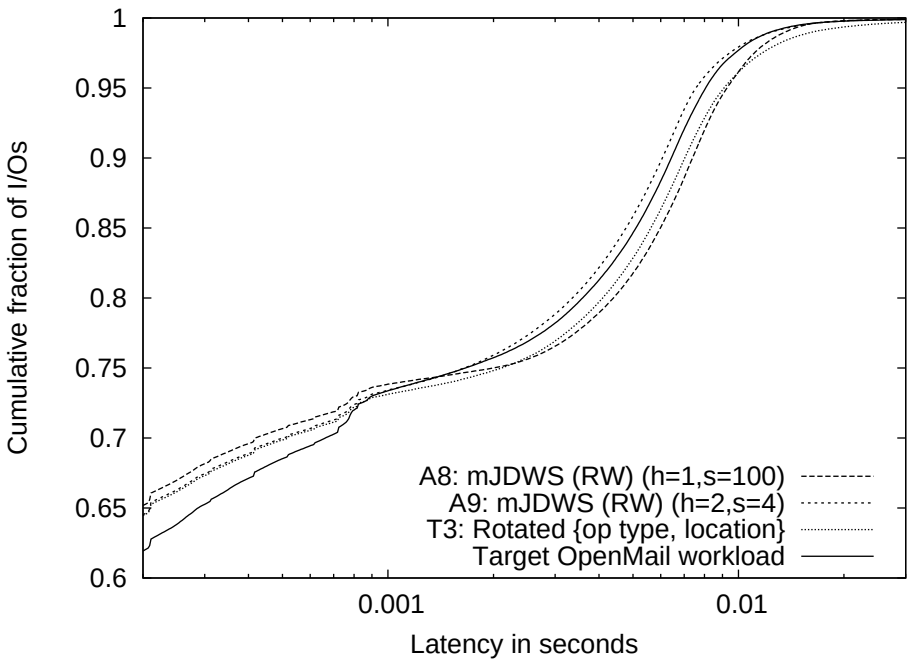

Figure 29: Comparison of {op. type, location} attributes chosen using RMS and log area demerit figures

Table 31: Comparison of {op. type, location} attributes chosen using RMS and log area

Internal demerit figure	Attribute	Attribute configuration	MRT	RMS	LA
RMS	$A8$	mJDWS (RW) ($h = 1$, $s = 100$)	-14%	14%	9%
LA	$A9$	mJDWS (RW) ($h = 2$, $s = 4$)	-35%	26%	6%

when synthesizing the OpenMail workload. When using the RMS demerit figure internally, the Distiller chooses mJDWS (RW) ($h = 1$, $s = 100$) (call it $A8$). When using the log area demerit figure, the Distiller chooses mJDWS (RW) ($h = 2$, $s = 4$) (call it $A9$). The Distiller produces the exploratory workload ($T3$) by rotating the operation type and location together.

Figure 29 shows that $A9$ matches $T3$ very closely when x is less than $.001s$, but that $A8$ matches $T3$ closely when x is greater than $.001s$. Both attributes have similar log area demerits; however, because RMS emphasizes differences for larger values of x, $A8$ has a smaller RMS demerit.

Figure 29 also shows the target workload. We see that when $x > .001s$, $A9$ follows

Table 32: Comparison of {op. type, location} attributes chosen for OLTP

Demerit figure	Attribute	Attribute configuration	MRT	RMS	LA
LA	$A10$	CD((op. type, loc.), loc., 2, 100)	20%	261%	24%
RMS	$A11$	mJDWS (RW) ($h = 1$, $s = 100$)	8%	23%	76%

the target workload more closely than $T3$. In other words, the errors in $A9$ offset the errors in $T3$. Consequently, adding $A9$ to the intermediate attribute list produces a very accurate intermediate synthetic workload.

A similar condition causes the synthetic OLTP workloads to be most accurate when we run the Distiller using the log area demerit figure internally. Figures 30 and 31 show the CDFs for the target OLTP workload, the rotated {operation type, location} workload, and the attributes the Distiller chose for the {operation type, location} location group using the RMS and log area demerit figures ($A10$ and $A11$). Figure 30 shows the entire CDF whereas Figure 31 shows only the tail of the distribution. Because the RMS demerit figure emphasizes tail behavior, $A11$ has a lower RMS demerit than $A10$. (Because we are evaluating individual attributes, we compare $A11$ and $A10$ to the rotated {operation type, location} workload. Table 32 lists the different demerits.) Because the log area demerit figure emphasizes differences at all time scales equally, $A10$ (with the smaller difference for the plateau between $x = .00007s$ and $x = .003s$) has a lower log area demerit than $A11$.

When the Distiller adds $A11$ to the intermediate attribute list and compares the resulting intermediate synthetic workload to the target OLTP workload, the intermediate workload's tail behavior more closely matches the rotated {operation type, location} workload than the target OLTP workload; consequently, the intermediate synthetic workload has a very high RMS error. In contrast, when the Distiller adds $A10$ to the intermediate attribute list, the CDF of the resulting synthetic workload has a tail that closely matches the target OLTP workload. Consequently, choosing attribute $A10$ produces lower log area and RMS demerits.

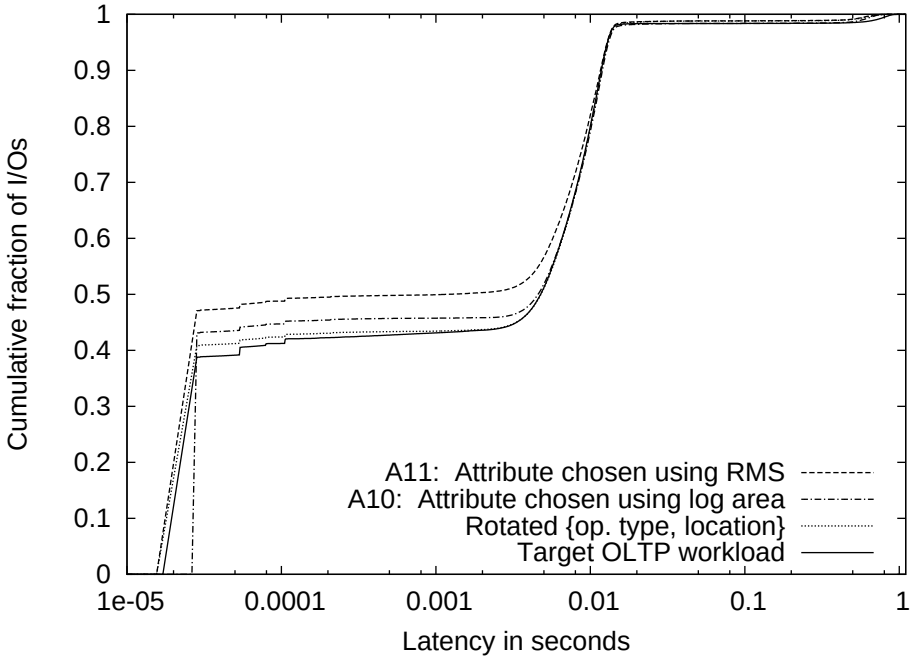

Figure 30: Comparison of {op. type, location} attributes chosen for OLTP

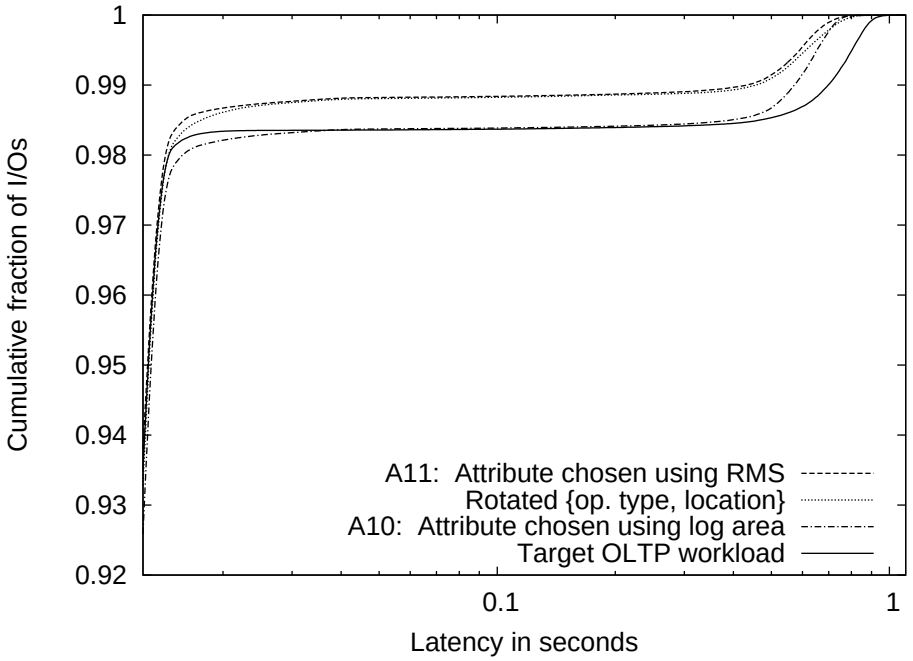

Figure 31: Comparison of {op. type, location} attributes chosen for OLTP (focus on tail)

Table 33: Counterintuitive log area results

Rotate amount	MRT	RMS	LA	Hybrid
25%	89%	13%	1%	89%
45%	30%	3%	2%	30%

Summary: We have seen two ways in which using the log area demerit figure internally can contribute to a final synthetic workload with a low RMS demerit figure. First, the log area demerit figure may emphasize differences between attributes that affect the RMS demerit figure only when evaluating an intermediate synthetic workload. Second, the attribute chosen using the log area demerit figure internally may have deficiencies that offset the deficiencies in the intermediate synthetic workload specified by the attributes chosen thus far.

We suspect that offsetting errors are possible given any pair of demerit figures. In fact, we suspect that offsetting errors may be common when using the MRT demerit figure because it represents only one data point (the mean response time); however, the effects of randomness error currently dominate the results of using the MRT and hybrid demerit figures. Therefore, we must first address the randomness error issues before studying how offsetting errors affect the Distiller when using the MRT and hybrid demerit figures.

7.1.3 Heavy tails

Response time distributions with extremely heavy tails (tails that stretch over several orders of magnitude) can produce very counterintuitive results. For example, Figure 25 and Table 29 provide one example of how a very heavy tail can produce a counterintuitive demerit: The RMS demerit for graphs Test 1 and Test 2 is 60%, despite the visual similarity and the low MRT and log area demerits. Fortunately, none of our current workload, storage system pairs produces such heavy tails. However, preliminary results show that we will face this problem when attempting to distill workloads using disk arrays with large prefetch lengths.

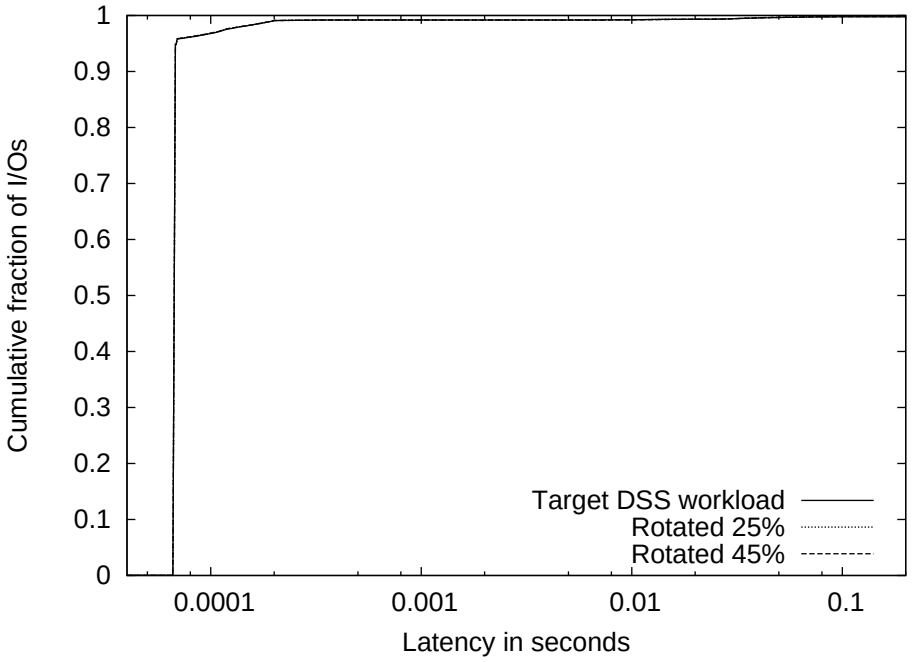

Figure 32: Counterintuitive log area results

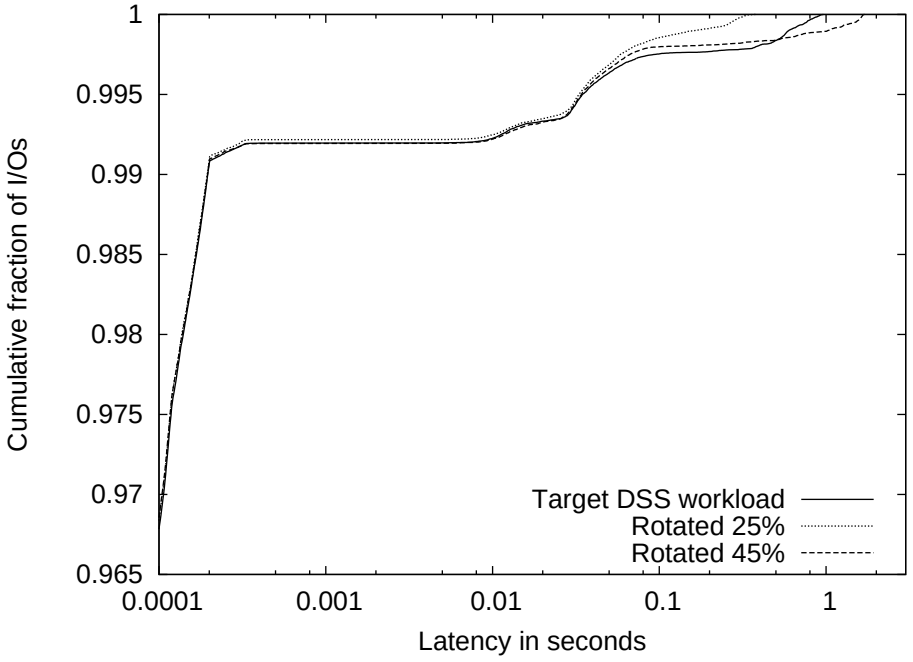

Figure 33: Counterintuitive log area results (focus on tail)

Table 26 shows how the use of prefetching increased the differences in performance between different rotated {location} workloads. When we examine individual rotated workloads for DSS using 512KB of prefetching, we see that workloads with low RMS or log area demerits can have very high MRT demerits. Very few people would consider a synthetic workload to be accurate if its mean response time differed from that of the target workload by 30%; however, we see in Figures 32 and 33 together with Table 33 that a workload with a 3% RMS demerit can have a 30% MRT demerit and a workload with a 1% log area demerit can have an 89% MRT demerit.

We are still searching for a good demerit figure with which to study the DSS workload when using prefetching. Because the MRT demerits are orders of magnitude larger than the log area demerits, the hybrid demerit figure degenerates into MRT. We do not expect the MRT demerit figure to accurately capture the long tail behavior. We have considered a combination of MRT and RMS; however, experience has shown that it is very difficult to produce a heavy-tailed synthetic workload with a low RMS demerit.

7.1.4 Lessons learned

The primarily limitation of the RMS demerit figure when used internally is that it considers absolute, as opposed to relative, differences between CDFs. By measuring absolute differences, the effects of a particular synthesis error on the RMS demerit depend on the time-scale of the I/Os in question. As we saw in Figures 27 and 28, the changes between the "internal" comparisons for individual attributes and "external" comparisons for intermediate synthetic workloads cause noticeable differences in how the synthesis errors of the attributes under test affect the RMS demerit and the Distiller's execution. Note, however, that this limitation does not affect the soundness of using RMS externally to evaluate the quality of the Distiller's final synthetic workload.

MRT is currently not a good internal demerit figure because the randomness errors and non-static behavior of our target workloads affect the MRT demerit figure enough to noticeably alter the distillation process. We believe we can overcome the randomness error problem by repeating evaluations to gain statistical confidence. We will not be able to study other potential limitations of MRT until we first address the randomness error limitation.

The log area demerit figure captures the overall differences between workloads more accurately than RMS and MRT; however, CDFs with similar shapes can have both a low log area demerit and a large MRT demerit (for example, OpenMail (All) and OpenMail (2GBLU)). To address this limitation, we developed the hybrid demerit figure. Our intent in taking the maximum of the log area and MRT demerit was to eliminate the situations where (1) the log area demerit was low but the workloads had very different mean response times, and (2) the MRT demerit was low but the workloads had very different behavior. Using the hybrid demerit had little effect on the OpenMail and DSS workloads, but had a large effect on the RMS and MRT metrics for the OLTP workload. This indicates that the hybrid demerit figure is potentially a better internal demerit figure than log area or MRT. We suspect that to fully understand the benefits of using the hybrid demerit figure, we must first address the underlying randomness error problems with MRT.

In summary, based on our observations, none of our demerit figures is clearly better than the others in all cases; however, the hybrid demerit figure leverages the benefits of the underlying log area demerit figure and leads the Distiller to near optimal final synthetic workloads for OLTP (All), DSS (All), and OpenMail (All).

In general, we expect the hybrid demerit figure to benefit the distillation of those workloads that are not prone to large randomness errors or non-static behaviors. We expect the log area metric to benefit the distillation of all workloads, except those whose response time distributions have tails so heavy that the MRT demerit can be

many times larger than the log area demerit. We expect the distillation of additional workloads to show that neither RMS nor MRT are good internal demerit figures for most workloads.

We do not expect that each disk array configuration will have its own optimal internal demerit figure. Instead, we expect that the choice of storage system will influence the choice of demerit figure primarily in the way it affects the target workload's distribution of response time. For example, we have found that replaying the DSS workload with large prefetch lengths produces workloads with extremely heavy tails, thereby limiting the effectiveness of the log area demerit figure.

7.2 Internal Threshold

In this section, we show that, as expected, raising the internal threshold reduces the Distiller's running time and the size of the attributes it chooses at a cost of reducing the quality of the final synthetic workloads. (Obtaining a synthetic workload's distribution of response time dominates the Distiller's running time. Therefore, we present running time in terms of the number of workloads generated and evaluated.) Using a first-fit criterion with a threshold of $x\%$ directs the Distiller to investigate any attribute groups for which the subtractive method shows a difference of more than $x\%$. After choosing an attribute group, the Distiller will select the first attribute with a demerit (relative to the attribute group) of at most $x\%$. Using a threshold of 0% causes the Distiller to evaluate all attribute groups and all attributes until an intermediate synthetic workload meets the stopping condition (the "external" threshold).

Table 34 presents the results of running the Distiller using four different internal thresholds. We studied only the OpenMail and OLTP workloads. We did not perform the demerit threshold tests on the DSS workload because is distilled using only thirteen simulations (one more than the minimum) and the final synthetic workloads have very low demerits. Tables 35 through 37 show the attributes chosen at each

Table 34: Effects of modifying internal threshold

Workload		Internal demerit	Thresh.	External demerit MRT	RMS	LA	Hybrid	Size	Test wklds.
OM	(All)	RMS	0%	6%	19%	14%	19%	40%	102/ 122
			7.5%	25%	26%	15%	26%	24%	64 / 90
			15%	19%	26%	27%	26%	25%	48 / 56
			25%	10%	29%	37%	37%	23%	19 / 25
OM	(All)	LA	0%	30%	47%	9%	30%	38%	93 / S
			7.5%	17%	9%	9%	17%	24%	27 / S
			15%	27%	35%	19%	35%	24%	22 / 22
			25%	23%	33%	23%	23%	24%	20 / 20
OLTP	(All)	LA	0%	11%	62%	25%	22%	67%	100/ 122
			7.5%	23%	145%	24%	24%	55%	51 / 60
			15%	22%	139%	24%	24%	55%	39 / 51
			25%	30%	187%	22%	30%	36%	25 / 25

iteration and the external demerits of each resulting intermediate synthetic workload.

RMS threshold for OpenMail: Table 34 shows that changing the RMS threshold has little effect on the RMS demerit of the final synthetic OpenMail workload. Changing the threshold does not have a significant effect on the attributes chosen until the threshold reaches 25%. Table 35 shows that the only significant improvements in the intermediate synthetic workloads come when the Distiller chooses an {operation type, location} attribute. Table 38 shows that the best {operation type, location} attribute, mJDWS (RW) ($h = 1$, $s = 100$, *pol*), has an RMS demerit of about 14%. In addition, all attributes with significantly smaller compact representations than mJDWS (RW) ($h = 1$, $s = 100$, *pol*) or mJDWS (RW) ($h = 1$, $s = 100$) have RMS demerits of at least 16%. (The smallest RMS demerits in Table 38 are in bold type.) Therefore, we must set the threshold to be greater than 16% in order for the Distiller to choose a different {operation type, location} attribute.

(Table 38 shows the order in which the Distiller evaluated {operation type, location} attributes when the RMS threshold was set to 0%. The order may differ slightly for other Distiller executions. When estimating an attribute's size, the Distiller compresses the attribute-value's Rome-formatted text representation using bzip

Table 35: Incremental results of distilling OpenMail with different internal RMS thresholds

Workload	Iter.	Attribute group	Error	Attribute added	Error	External demerit MRT	RMS	LA
OM (All)						46%	95%	139%
RMS 0%	1	{loc.}	20%	mJDWS($s = 1$, $s = 100$)	8%	54%	100%	144%
	2	{iat}	17%	Cluster($ws = 10.24$, $bs = .0025$, $c = .3$)	3%	53%	96%	123%
	3	{op. type}	4%	Operation type STC	4%	47%	95%	122%
	4	{req. size}	3%	CD(req. size, req. size, 2, 10)	1%	51%	94%	123%
	5	{op. type, loc.}	72%	mJDWS (RW) ($h = 1$, $s = 100$, *pol*)	14%	25%	**26%**	**14%**
	6	{op. type, iat}	11%	Cluster($ws = 10.24$, $bs = .01$, $c = .3$)	9%	19%	26%	14%
	7	{op. type, req. size}	4%	Joint dist. of op. type and req. size	4%	**6%**	26%	21%
OM (All)						49%	96%	140%
RMS 7.5%	1	{loc.}	20%	mJDWS($s = 1$, $s = 100$)	8%	54%	100%	144%
	2	{iat}	17%	β-model ($ws = 5.12$, $il = .01$, $s = 10^{-2}$)	3%	44%	97%	155%
	3	{op. type, loc.}	72%	mJDWS (RW) ($h = 1$, $s = 100$, *pol*)	14%	24%	**26%**	30%
	4	{op. type, iat}	11%	Cluster($ws = 10.24$, $bs = .01$, $c = .3$)	9%	25%	26%	**15%**
	5	{loc., req. size}	30%	CD(loc., req. size, 1, 100)	23%	**20%**	26%	17%
OM (All)						46%	96%	139%
RMS 15%	1	{loc.}	20%	mJDWS($h = 1$, $s = 10$)	8%	56%	100%	144%
	2	{iat}	17%	β-model ($ws = 5.12$, $il = .01$, $s = 10^{-2}$)	3%	53%	98%	156%
	3	{op. type, loc.}	72%	mJDWS (RW) ($h = 1$, $s = 100$)	14%	25%	**26%**	28%
	4	{loc., req. size}	30%	CD(loc., req. size, 1, 100)	23%	**19%**	27%	**27%**
OM (All)						44%	96%	140%
RMS 25%	1	{op. type, loc.}	72%	Run count within state ($h = 2$, $s = 4$)	24%	**10%**	**29%**	38%
	2	{loc., req. size}	30%	CD(loc., request size, 1, 4)	23%	12%	29%	**37%**

Table 36: Incremental results of distilling OpenMail with different internal log area thresholds

						External demerit		
Workload	**Iter.**	**Attribute group**	**Error**	**Attribute added**	**Error**	**MRT**	**RMS**	**LA**
OM (All)						47%	96%	138%
Log area 0%	1	{iat}	21%	Cluster($ws = 2.56$, $bs = .0025$, $c = .3$)	0%	48%	92%	121%
	2	{req. size}	4%	CD(req. size, req. size, 3, 4)	1%	44%	91%	122%
	3	{location}	3%	JDWS($h = 1$, $s = 100$)	2%	43%	95%	125%
	4	{op. type}	1%	CD(op. type, op. type, 8, 2)	0%	46%	93%	123%
	5	{op. type, loc.}	137%	mJDWS (RW) ($h = 1$, $s = 1$)	5%	**30%**	**47%**	**9%**
OM (All)						46%	95%	136%
Log area 7.5%	1	{iat}	20%	β-model ($ws = 5.12$, $il = .01$, $s = 10^{-5}$)	8%	43%	91%	137%
	2	{op. type, loc.}	137%	mJDWS (RW) ($h = 2$, $s = 4$)	6%	**17%**	**9%**	**9%**
OM (All)						47%	96%	138%
Log Area 15%	1	{iat}	20%	β-model ($ws = 5.12$, $il = .01$, $s = 10^{-3}$)	13%	52%	91%	142%
	2	{op. type, loc.}	137%	CD(op. type, location, 1, 2)	12%	**27%**	**35%**	**19%**
OM (All)						47%	96%	138%
Log Area 25%	1	{op. type, loc.}	137%	CD(op. type, location, 1, 2)	12%	**23%**	**33%**	**23%**

Table 37: Incremental results of distilling OLTP with different thresholds

Workload	Iter.	Attribute group	Error	Attribute added	Error	External demerit MRT	RMS	LA
OLTP (All)						39%	182%	197%
Log area 0%	1	{iat}	25%	Cluster ($ws = 5.12$, $bs = .01$, $c = .3$)	1%	42%	204%	170%
	2	{loc.}	5%	CD(loc., loc., $h = 2$, $s = 1$)	5%	42%	204%	170%
	3	{op type}	2%	Read percentage	2%	42%	204%	170%
	4	{req. size}	1%	CD(req. size, req. size, $h = 3$, $s = 4$)	0%	44%	217%	158%
	5	{op. type, loc.}	174%	CD((op. type, loc.), loc., 2, 100)	24%	35%	242%	25%
	6	{op. type, req. size}	4%	CD(op. type, req. size, $h = 1$, $s = 2$)	1%	**11%**	62%	29%
	7	{op. type, iat}	3%	Cluster($ws = 2.56$, $bs = .01$, $c = .3$)	1%	12%	**65%**	**22%**
	8	{loc., iat}	20%	CD(loc., iat, $h = 1$, $s = 100$)	20%	15%	82%	22%
	9	{loc., req. size}	5%	CD(loc., req. size, 2, 10)	8%	26%	174%	43%
OLTP (All)						40%	184%	200%
Log area 7.5%	1	{op type}	35%	Read percentage	6%	41%	195%	154%
	2	{op. type, loc.}	174%	CD((op. type, loc.), loc., 2, 100)	30%	**23%**	**145%**	32%
	3	{loc., iat}	20%	CD(loc., iat, 3, 2)	21%	24%	155%	**24%**
OLTP (All)						44%	216%	155%
Log area 15%	1	{iat}	25%	IAT STC (.001)	1%	47%	238%	164%
	2	{op. type, loc.}	174%	CD((op. type, loc.), loc., 2, 100)	24%	**22%**	**139%**	**24%**
	3	{loc., iat}	20%	CD(loc., iat,$h = 1$, $s = 100$)	20%	30%	207%	25%
OLTP (All)						47%	237%	171%
Log area 25%	1	{iat}	25%	IAT STC (.001)	1%	40%	**187%**	180%
	2	{op. type, loc.}	174%	CD(op. type, loc., 1, 2)	24%	**30%**	209%	**22%**

Table 38: Size and demerit of {operation type, location} attributes when distilling OpenMail

Attribute	Size	Internal Demerit figure MRT	RMS	LA	Hybrid
mRCWS (RW) ($h = 1$, $s = 20$)	893796	37%	44%	77%	77%
mRCWS (RW) ($h = 2$, $s = 8$)	903605	24%	21%	35%	35%
mRCWS (RW) ($h = 1$, $s = 2$)	911873	47%	44%	76%	76%
mJDWS (RW) ($h = 1$, $s = 10$)	944926	31%	23%	9%	31%
mJDWS (RW) ($h = 1$, $s = 10$, *pol*)	944961	27%	25%	13%	27%
mJDWS (RW) ($h = 2$, $s = 4$)	967284	35%	26%	6%	35%
mJDWS (RW) ($h = 2$, $s = 4$, *pol*)	967287	35%	27%	7%	35%
CD(Op. type, location, 1, 1)	968866	7%	18%	11%	11%
mJDWS (RW) ($h = 1$, $s = 1$)	969804	14%	**16%**	5%	14%
mJDWS (RW) ($h = 1$, $s = 1$, *pol*)	969957	12%	40%	5%	12%
mJDWS (RW) ($h = 2$, $s = 10$)	970809	31%	26%	11%	31%
mJDWS (RW) ($h = 2$, $s = 10$, *pol*)	970809	29%	23%	12%	29%
mRCWS (RW) ($h = 2$, $s = 100$)	1009300	27%	20%	36%	36%
mJDWS (RW) ($h = 1$, $s = 100$, *pol*)	1030481	8%	**14%**	13%	13%
mJDWS (RW) ($h = 1$, $s = 100$)	1030511	13%	**14%**	9%	13%
mRCWS (RW) ($h = 1$, $s = 200$)	1047510	50%	47%	82%	82%
CD(Op. type, location, 1, 100)	1737953	3%	15%	15%	15%

and compares the size of the compressed file to other compressed attribute-values. The combination of the Rome format and bzip can produce small differences in the order in which the Distiller evaluates attributes. The perl implementation of the Rome library uses associative arrays. When iterating through an associative array, perl makes no guarantees about the order in which keys are returned. This slight difference in order can affect bzip's compression ratio.)

The log area demerit of the final synthetic workload increases when we raise the threshold from 7.5% to 15% because the Distiller explores the {operation type, interarrival time} attribute group only when we set the threshold below 11%. When the threshold is 7.5%, the addition of the IAT clustering attribute in iteration 4 reduces the log area demerit from 28% to 15%. When the threshold is 15%, the Distiller does not explore the {operation type, interarrival time} attribute group and the log area demerit remains at 28%.

Changing the threshold does, however, have a significant effect on the number of

Table 39: Size and demerit of {interarrival time} attributes when distilling OpenMail

Attribute	Size	Internal Demerit figure MRT	RMS	LA	Hybrid
β-model($ws = 5.12$, $il = 5.12$, $s = .01$)	2005	2%	**3%**	20%	20%
β-model($ws = 5.12$, $il = 5.12$, $s = .001$)	2007	1%	6%	13%	13%
β-model($ws = 5.12$, $il = 5.12$, $s = .00001$)	2010	2%	5%	9%	9%
IAT STC (.001)	2171	6%	10%	15%	15%
IAT STC (.001,.002)	2243	6%	11%	15%	15%
IAT STC (.002)	2257	6%	11%	16%	16%
CD(iat, iat, 3, 4)	23571	5%	9%	11%	11%
CD(iat, iat, 4, 3)	27494	5%	8%	11%	11%
CD(iat, iat, 2, 10)	38171	6%	10%	12%	12%
CD(iat, iat, 1, 100)	33053	7%	13%	14%	14%
IAT cluster($ws = 5.12$, $bs = .01$, $c = .3$)	551909	1%	3%	1%	1%
IAT cluster($ws = 5.12$, $bs = .0025$, $c = .3$)	551909	1%	1%	1%	1%
IAT cluster($ws = 2.56$, $bs = .0025$, $c = .3$)	555561	1%	3%	0%	1%
IAT cluster($ws = 2.56$, $bs = .01$, $c = .3$)	556673	1%	2%	1%	1%
IAT cluster($ws = 10.24$, $bs = .0025$, $c = .3$)	556706	0%	**1%**	1%	1%
IAT cluster($ws = 10.24$, $bs = .01$, $c = .3$)	567487	1%	3%	3%	3%

exploratory workloads the Distiller generates and evaluates (which dominates the running time), and on size of the final synthetic workloads' compact representations. Table 35 shows that increasing the threshold causes the Distiller to explore fewer attribute groups and, hence, generate and evaluate fewer exploratory workloads. Furthermore, increasing the threshold reduces the number of attributes the Distiller evaluates when exploring an attribute group. For example, when we set the threshold to 0%, the Distiller evaluates every {interarrival time} attribute, even though the first attribute it evaluates has a demerit of only 3%. (See Table 39.) The complete exploration of this attribute group alone adds 25 evaluations to the Distiller's running time. Similarly, Table 38 shows that raising the threshold from 7.5% to 15% prevents the Distiller from continuing to evaluate {operation type, location} attributes after it finds that mJDWS (RW) ($h = 1$, $s = 100$) has an RMS demerit of 14%. In contrast, with a 7.5% threshold, the Distiller evaluates every {operation type, location} attribute.

Changing the threshold affects the size of the final synthetic workloads' compact

representations in two ways: First, when the Distiller does not explore an attribute group, it does not choose an attribute from that group whose value adds to the size of the compact representation. Second, because the Distiller evaluates the most compactly-represented attributes first, raising the thresholds causes the Distiller to choose more compactly represented attributes. For example, Table 39 shows the order in which OpenMail evaluates {interarrival time} attributes and highlights the attributes chosen given different internal thresholds. When we set the threshold to 0%, the Distiller chooses an IAT clustering {interarrival time} attribute. However, when we raise the threshold, the Distiller instead chooses a much more compact β-model. This change alone reduces the size of the final synthetic workload's compact representation from 40% to 25% of the target workload's size.

Log area threshold for OpenMail: Increasing the log area threshold for OpenMail also has the expected effect of raising the log area demerit of the final synthetic workload while reducing the size of the final synthetic workload's compact representation and the Distiller's running time. Offsetting errors and randomness errors prevent the RMS and MRT demerit figures from strictly increasing as the log area threshold increases.

Log area threshold for OLTP: Changing the threshold has less effect on the OLTP workload than on the OpenMail workload. We still see reductions in the number of workloads the Distiller evaluates and the size of the final synthetic workload's compact representation; however, these changes are smaller because the attribute library does not describe the OLTP workload as well as it describes OpenMail. Therefore, the Distiller must still evaluate more and larger attributes, even when using higher thresholds. The only significant effect on the quality of the final synthetic workload is that setting the threshold to 0% allows the Distiller to investigate the {operation type, request size} group. Adding a conditional distribution of request size based on operation type significantly reduces the RMS demerit of the final synthetic

workload.

Our experience shows that setting the threshold to 7.5% works well for producing a final synthetic workload with a 10% demerit — provided the attribute library is sufficient. When the attribute library is not sufficient, we believe the threshold should be set slightly higher than the demerit of the most accurate attribute in each group. The challenge is that we do not know *a priori* the demerit of the best attribute in each group. Therefore, we suspect that the threshold will have to be chosen based on user experience. Thus far, our experience with 7.5% has been positive.

7.3 Size / Accuracy tradeoff

Table 16 in Chapter 6 shows that the Distiller can find reasonably accurate synthetic workloads. Unfortunately, the defining attributes of these synthetic workloads tend to have large representations. We lose the size benefit of using a synthetic workload when its compact representation is nearly as large as the workload trace itself. Fortunately, we can trade accuracy for conciseness of representation when generating synthetic workloads. In Section 7.2, we saw how we could implicitly trade compact representation size for accuracy by raising the Distiller's threshold for investigating attribute groups and choosing key attributes. In this section, we show how we can explicitly trade size for accuracy by reducing the precision of the attributes the Distiller uses to specify the final synthetic workload. Specifically, we show that we can reduce the size of the synthetic OpenMail workload's compact representation by 70% without increasing either the log area or RMS demerits. As a result of offsetting errors, reducing the size of the synthetic OLTP workload by 5% actually doubles the accuracy. Further reductions of about 20% increase the RMS and log area demerits by about 20%.

We can measure and present many attributes with varying degrees of precision.

For example, the number of bins used to represent a distribution determines its precision — histograms with more bins more accurately represent the distribution. Similarly, a conditional distribution's accuracy tends to increase as the number of states increases. To leverage this tradeoff, we run the Distiller in "post-process" mode, where it varies the precisions of the attributes that specify the final synthetic workload. Sections 7.3.1 and 7.3.2 explore this tradeoff for the OpenMail and OLTP workloads respectively. We do not present results for the DSS workload because the Distiller chose very accurate and compact attributes during its initial execution.

7.3.1 OpenMail

We first analyze the effects of post-processing the set of attributes the Distiller chose for OpenMail (using the log area demerit figure internally). For this workload, the Distiller chose the β-model with parameters (ws = 5.12, bs = .01, s = 10^{-5}) to describe the arrival pattern and mJDWS (RW) with parameters (h = 2, s = 4) to describe the access pattern. The Distiller also chose a transition matrix to describe the sequence of operation types (mJDWS (RW) implicitly generates operation type using its transition matrix) and the empirical distribution for request sizes (the default for request size). Table 40 provides the details of OpenMail's distillation.

Among the chosen attributes, only mJDWS (which requires distributions of location and modified jump distance for each defined state) can be measured with significantly differing degrees of precision. The bins for the distribution of location can be as small as 1KB (one sector) or as large as the address space of the workload. When the bins are 1KB, the histogram for location contains 162196 bins. The other chosen attributes use histograms with at most 128 bins. The potential space savings from changing these bin widths is negligible compared to the histograms for location. Therefore, we focus on only the number of states and bin widths for the location histograms.

Table 40: Incremental results of distilling OpenMail (All)

Workload	Iter.	Attribute group	Error	Attribute added	Error	External demerit MRT	RMS	LA
OM (All)						46%	95%	136%
Log area 7.5%	1	{iat}	20%	β-model ($ws = 5.12$, $il = .01$, $s = 10^{-5}$)	8%	43%	91%	137%
	2	{op. type, loc.}	137%	mJDWS (RW) ($h = 2$, $s = 4$)	6%	**17%**	**9%**	**9%**

Table 41: RMS demerit and size for OpenMail attributes of varying precision

Bin width	mJDWS states (history, number of states)									
(sectors)	(1,1)		(2,4)		(1,10)		(2,10)		(1,100)	
1	90%	(.24)	9%	(.24)	10%	(.24)	6%	(.25)	29%	(.26)
8	110%	(.21)	13%	(.21)	7%	(.21)	8%	(.22)	30%	(.24)
64	108%	(.14)	16%	(.14)	7%	(.14)	9%	(.15)	36%	(.17)
1024	114%	(.07)	21%	(.07)	7%	(.07)	9%	(.08)	43%	(.08)
10240	194%	(.02)	22%	(.03)	14%	(.03)	18%	(.03)	48%	(.05)
25600	153%	(.01)	40%	(.01)	18%	(.02)	27%	(.02)	Err	(.04)
102400	194%	(.01)	201%	(.01)	28%	(.01)	28%	(.01)	48%	(.03)

Table 42: Log area demerit and size for OpenMail attributes of varying precision

Bin width	mJDWS states (history, number of states)									
(sectors)	(1,1)		(2,4)		(1,10)		(2,10)		(1,100)	
1	10%	(.24)	10%	(.24)	11%	(.24)	11%	(.25)	14%	(.26)
8	12%	(.21)	12%	(.21)	11%	(.21)	12%	(.22)	15%	(.24)
64	12%	(.14)	11%	(.14)	10%	(.14)	10%	(.15)	15%	(.17)
1024	12%	(.07)	10%	(.07)	10%	(.07)	9%	(.08)	16%	(.08)
10240	13%	(.02)	10%	(.03)	10%	(.03)	11%	(.03)	17%	(.05)
25600	13%	(.01)	10%	(.01)	11%	(.02)	11%	(.02)	Err	(.04)
102400	13%	(.01)	11%	(.01)	11%	(.01)	11%	(.01)	17%	(.03)

Table 43: MRT demerit and size for OpenMail attributes of varying precision

Bin width	mJDWS states (history, number of states)									
(sectors)	(1,1)		(2,4)		(1,10)		(2,10)		(1,100)	
1	50%	(.24)	18%	(.24)	17%	(.24)	18%	(.25)	23%	(.26)
8	49%	(.21)	19%	(.21)	15%	(.21)	2%	(.22)	18%	(.24)
64	49%	(.14)	21%	(.14)	22%	(.14)	29%	(.15)	42%	(.17)
1024	52%	(.07)	31%	(.07)	31%	(.07)	35%	(.08)	49%	(.08)
10240	74%	(.02)	45%	(.03)	50%	(.03)	46%	(.03)	60%	(.05)
25600	70%	(.01)	61%	(.01)	48%	(.02)	56%	(.02)	Err	(.04)
102400	70%	(.01)	66%	(.01)	57%	(.01)	50%	(.01)	58%	(.03)

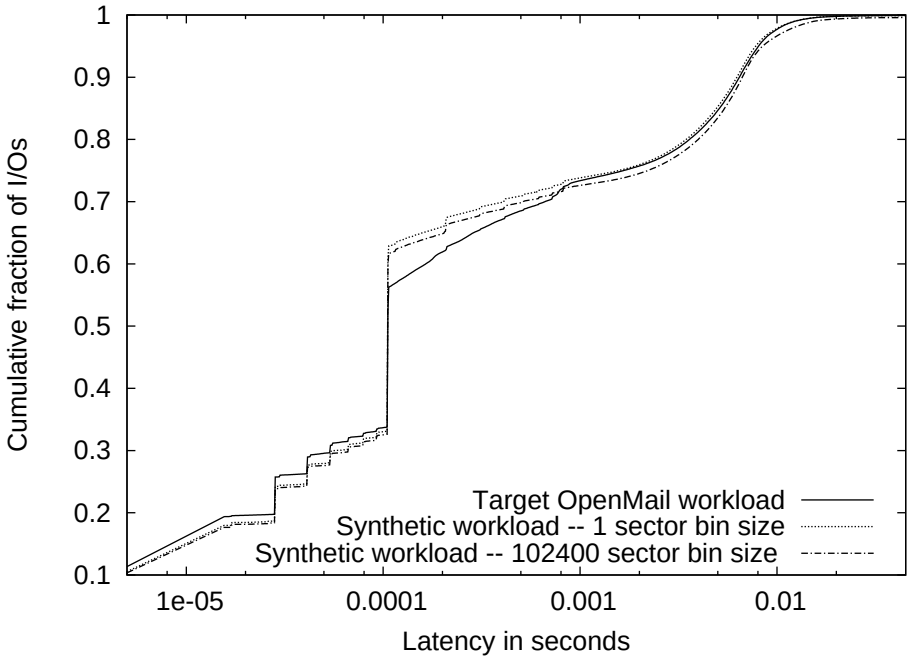

Figure 34: Effects of increasing bin width on OpenMail workload

Tables 41 through 43 show the external demerits of synthetic workloads produced using five different state definitions and seven different bin widths. The number in parentheses next to each demerit is the size of the synthetic workload's compact representation relative to the size of the compressed target workload trace.[1]

Effects of increasing bin width: When looking down each column in Tables 41 through 43, we see that increasing the bin width from 1 to 102400

- causes the RMS demerit to increase up to a factor of 25,
- has little effect on the log area demerit, and
- causes the MRT demerit to increase up to a factor of 4.

We can explain these changes by examining (1) how increasing the bin width affects

[1]Generating a sequence of locations that match both a given distribution of location and a given distribution of jump distance is NP-complete. Therefore, our generation techniques use an approximation algorithm (presented in Appendix A). Unfortunately, this approximation algorithm fails in the case where $h = 1$, $s = 100$, and the bin width is 25600 sectors.

the synthetic workload's footprint and (2) how the resulting change in response time distribution affects the demerit figures.

Increasing the bin width tends to reduce the amount of locality in a workload. When a synthetic workload generator draws locations from a distribution based on a histogram with bins larger than one sector, it must make an assumption about the distribution of locations within each bin. Our random number generation techniques assume uniform distributions within bins. This assumption is not true in general for the OpenMail workload. Other than the first few hundred thousand sectors, most locations tend to be multiples of 8KB. As a result, drawing locations from a histogram with large bins produces a workload with a larger footprint and less locality than the target workload. For example, the requests for the target OpenMail workload use 162196 unique location values, whereas the synthetic workload generated using 25600 sector bins uses 489584 unique location values. (The number of location values approximates the footprint size.) The larger footprint and decreased locality tend to produce a slower workload.

The CDFs of slower workloads tend to have flatter tails and lie to the right of the CDFs for faster workloads. Figure 34 provides an example by showing the CDFs of response time for two synthetic workloads. We generated both workloads using JDWS (RW) ($h = 1$, $s = 10$); however, the faster workload uses a 1 sector bin width and the slower workload uses a 102400 sector bin width.

The log area demerit figure is more sensitive to changes in a CDF's shape than to slight changes in position. Figure 34 shows that the different synthetic workloads have similar shapes; therefore, as observed in Tables 41 through 43, increasing the bin width has a small effect on the log area demerit. In contrast, the RMS demerit figure is more sensitive to shifts in position, especially shifts in the tail; therefore, the RMS demerit figure increases as bin width increases. Finally, because (1) increasing the bin width tends to slow the resulting synthetic workload, and (2) the synthetic OpenMail

workloads have larger mean response times than the target OpenMail workload, the MRT demerit also tends to increase as bin width increases.

Increasing the number of states: When looking across each row, we see that increasing the number of states into which we divide location does not necessarily produce a more accurate synthetic workload. Specifically, increasing the number of states from 10 to 100 increases the values of all four of our demerit figures. The primary motivation for the jump distance within state attribute is to measure jump distance from the perspective of individual disks. Therefore, a jump distance within state attribute should be most effective when we define the states to correspond roughly to the storage system's physical disks. The challenge is that RAID configurations and the consequent striping of data make the correspondence between states and physical disks approximate. The OpenMail workload comprises 22 LUs; each LU is mapped onto a RAID group consisting of four physical disks. Thus, of the state configurations tested, those that divide the address space into ten regions best match the physical configuration of the simulated disk array. Future work includes configuring an mJDWS attribute with states that correspond to individual LUs and evaluating whether it defines a more accurate synthetic workload than mJDWS configured using the percentile method.

For OpenMail, the state configurations that use ten location regions are also the configurations least affected by increasing the bin width. When using mJDWS(RW) ($h = 1$, $s = 10$) and the RMS demerit figure, we can reduce the synthetic workload's compact representation by about 70% without degrading the quality of the synthetic workload. When using the log area demerit figure, we can reduce the compact representation by 98%. Because the mean response time increases as bin width increases, reducing the size of the compact representation increases the MRT and hybrid demerits.

Overall tradeoff: Figure 35 combines the data presented in Tables 41 through 43;

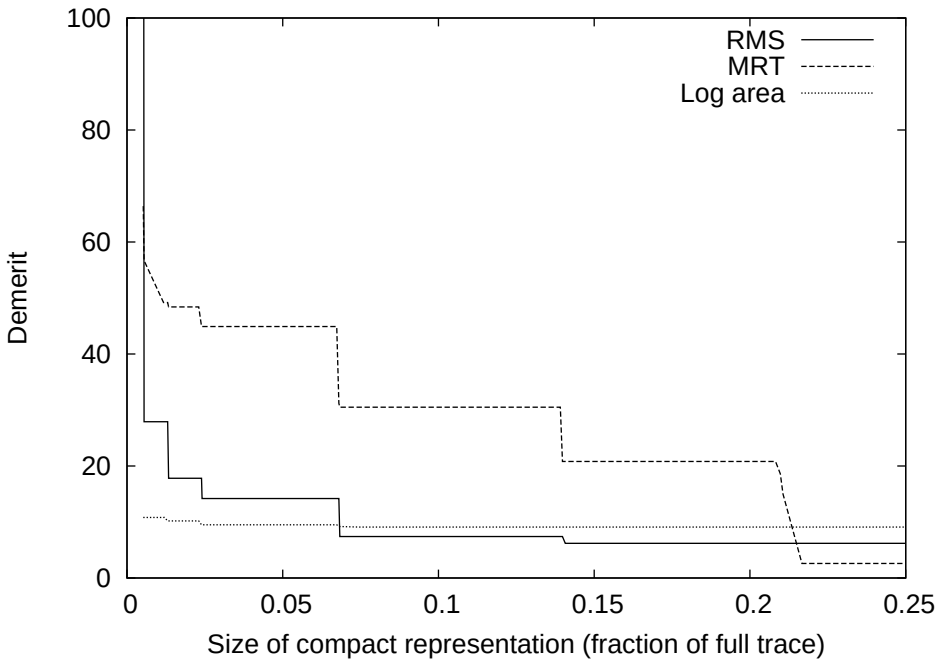

Figure 35: OpenMail size / accuracy tradeoff

it illustrates the tradeoff between the size of the attributes and the accuracy of the resulting final synthetic workload. Each y value is the external demerit of the most accurate synthetic workload with a compact representation no larger than $x\%$ of the compressed target workload's size. Thus, if we stipulate that a workload's compact representation may be no larger than 10% of the size of the compressed original workload trace, then the best synthetic workload has an external log area demerit of 7%. The "best" workloads will produce points in the lower-left corner of the graph — these workloads will both be accurate and have small compact representations.

As seen in Tables 41 and 43, the RMS and MRT demerits improve steadily as size increases. Decreasing the bin width both raises the compact representation size and moves the tail of the resulting CDF closer to that of the target workload, thereby reducing the RMS demerit. Decreasing the bin width also reduces the size of the resulting synthetic workload's footprint, thereby decreasing the mean response time and the MRT demerit. Table 42 shows that the log area demerit figure remains steady at around 10%. The resulting decrease in footprint size from decreasing the bin width

has little effect on the shape of the resulting synthetic workloads; and, consequently, has little effect on the log area demerit figure.

7.3.2 OLTP

This section analyzes the effects of post-processing the set of attributes the Distiller chose for the OLTP workload. Table 44 lists the attributes the Distiller chose during each iteration. For the final synthetic workload, the arrival pattern, the sequence of request sizes, and the sequence of operations types are chosen independently at random from the observed distributions. The sequence of locations is generated from a conditional distribution that depends on the current operation type and both the operation type and location of the previous I/O: CD((op. type, location), location, 2, 100).

Tables 45 through 47 show the effects of modifying the state definition and bin width for the OLTP workload. When specifying state definitions, the first number (h) refers to the number of requests considered and the second number (s) refers to the number of regions into which we divide the address space. When $h = 1$, the attribute considers only the operation type of the current request. Therefore, unlike mJDWS(RW) used for OpenMail, the conditional distribution used for OLTP has only one configuration when $h = 1$. In addition, the limitations of our approximation algorithm for choosing a sequence of locations based on distributions of location and jump distance prevent us from studying bin widths larger than 24576 for the OLTP workload.

Like OpenMail, the effects of modifying the attributes for OLTP are small when using the log area demerit figure and larger when using the RMS and MRT demerit figures. The main difference between OLTP and OpenMail is that the synthetic OLTP workloads tend to be more accurate when using an 8 sector bin width than when using a 1 sector bin width. This increase in accuracy is the result of offsetting errors. Although the chosen {operation type, location} attribute is the best available,

Table 44: Incremental results of distilling OLTP (All)

Workload	Iter.	Attribute group	Error	Attribute added	Error	External demerit MRT	RMS	LA
OLTP (All)						40%	184%	200%
Log area 7.5%	1	{op type}	35%	Read percentage	6%	41%	195%	154%
	2	{op. type, loc.}	174%	CD((op. type, loc.), loc., 2, 100)	30%	**23%**	**145%**	32%
	3	{loc., iat}	20%	CD(loc., iat, 3, 2)	21%	24%	155%	**24%**

Table 45: RMS demerit and size for OLTP attributes of varying precision

Bin width (sectors)	Conditional distribution states (history, number of states) (1,1)		(2, 50)		(2,100)	
1	170%	(.37)	165%	(.54)	170%	(.55)
8	78%	(.26)	81%	(.49)	88%	(.49)
64	97%	(.09)	86%	(.38)	97%	(.40)
1024	128%	(.01)	81%	(.14)	98%	(.20)
24576	156%	(.004)	100%	(.01)	109%	(.03)

Table 46: Log area demerit and size for OLTP attributes of varying precision

Bin width (sectors)	Conditional distribution states (history, number of states) (1,1)		(2, 50)		(2,100)	
1	33%	(.37)	31%	(.54)	24%	(.55)
8	13%	(.26)	18%	(.49)	16%	(.49)
64	13%	(.09)	13%	(.38)	20%	(.40)
1024	13%	(.01)	13%	(.14)	21%	(.20)
24576	16%	(.004)	13%	(.01)	13%	(.03)

Table 47: MRT demerit and size for OLTP attributes of varying precision

Bin width (sectors)	Conditional distribution states (history, number of states) (1,1)		(2, 50)		(2,100)	
1	25%	(.37)	24%	(.54)	25%	(.55)
8	13%	(.26)	12%	(.49)	15%	(.49)
64	10%	(.09)	13%	(.38)	10%	(.40)
1024	4%	(.01)	12%	(.14)	9%	(.20)
24576	0%	(.004)	7%	(.01)	6%	(.03)

it specifies a workload with too much locality. Increasing the bin width offsets this error by reducing the amount of locality.

Figure 36 shows the performance of two synthetic OLTP workloads generated with the (h = 2,s = 100) state configurations. We can see that the synthetic workload specified by the Distiller (bin width 1) has too much locality because there are too many cache hits (i.e., I/Os with $3e^{-5}$s response times). We can also see that the synthetic workload specified using an 8 sector bin size has fewer cache hits (because it has fewer $3e^{-5}$s I/Os). Thus, the reduction in locality produced a synthetic workload whose performance is closer to the target than the performance of a synthetic workload generated using a 1 sector bin width.

Figure 37 shows the tails of the CDFs of three synthetic OLTP workloads generated with the (h = 2,s = 100) state configurations and varying bin widths. We can see that the tail behavior of the synthetic workload with an 8 sector bin width is much closer to that of the target OLTP workload than the workloads with 1 and 24576 sector bin widths. The same effects on locality that lower the plateau between $x = 3e^{-5}$s and x = .002s also lower the plateau between x = .015s and x = .4s. This difference in tail behavior causes the RMS demerit figure to decrease when the bin width increases from 1 to 8.

As with the OpenMail workload, increasing the bin width also increases the mean response time. However, unlike the synthetic OpenMail workloads, the synthetic OLTP workloads are faster than the target workload (i.e., the synthetic workloads have lower mean response times than the target). As a result, the increase in mean response time produces a decrease in the MRT demerit (because the MRT demerit figure is an absolute value).

Figure 38 illustrates the tradeoff between size and accuracy for the OLTP workload. On the left, we can see that the OLTP workload improves from 100% RMS demerit to 80% RMS demerit when we reduce the bin width from 24567 to 1024.

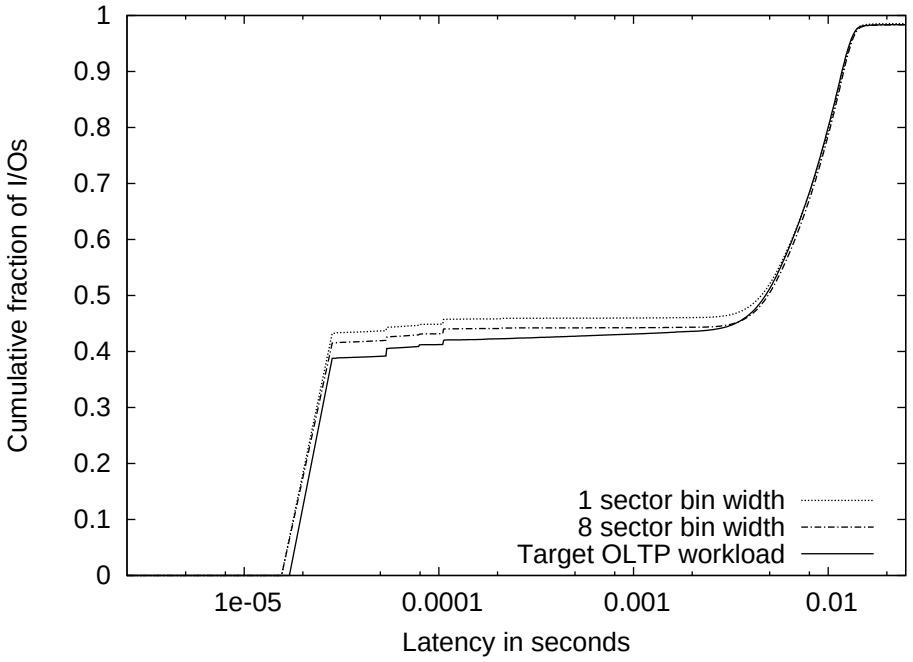

Figure 36: Example of OLTP tradeoff

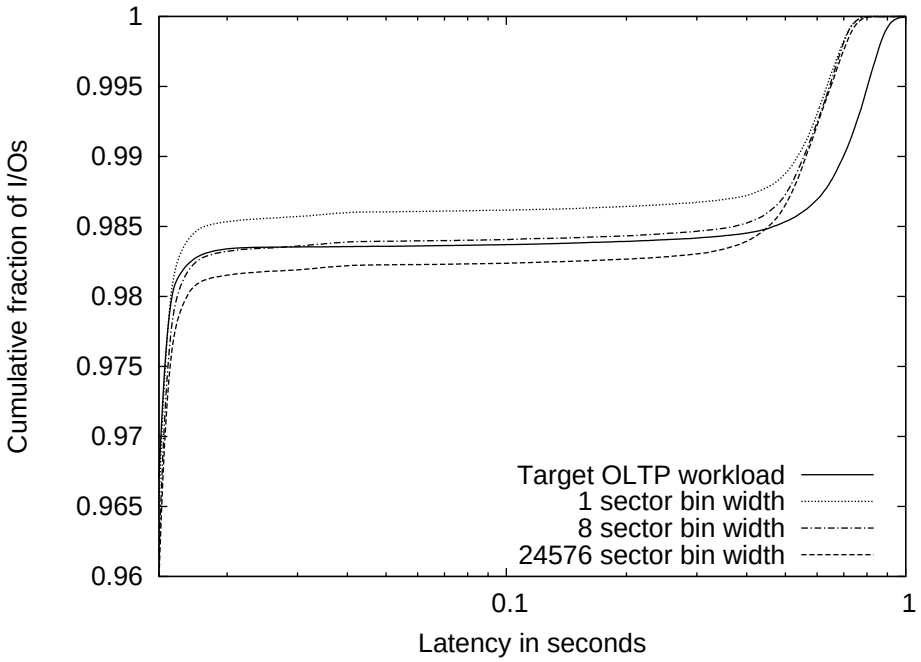

Figure 37: Example of OLTP tradeoff (focus on tail)

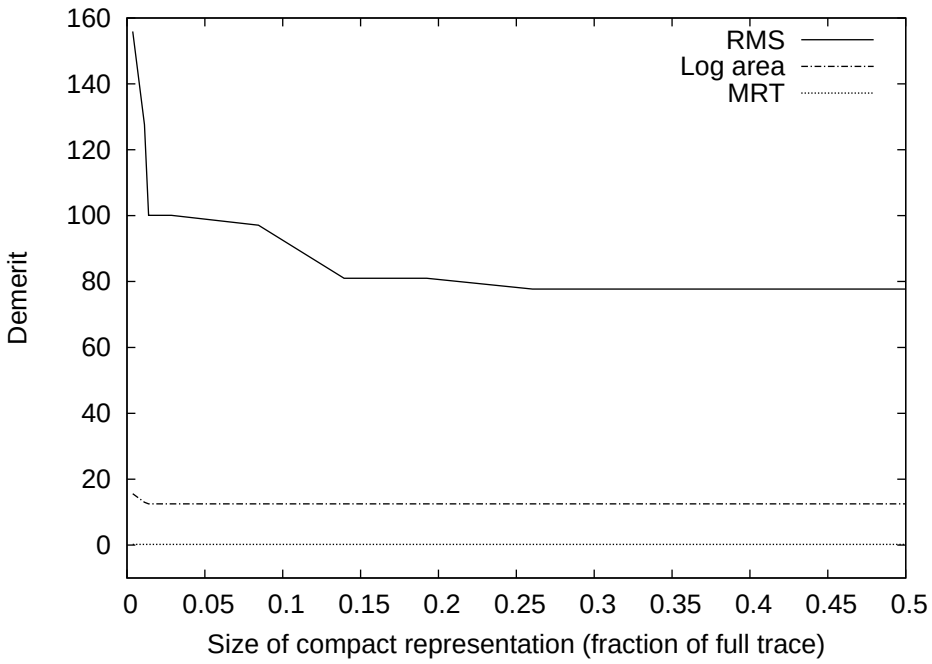

Figure 38: OLTP size / accuracy tradeoff

Furthermore, we see that the RMS demerit does not fall below 80%. As shown in Table 46, the log area demerits remain nearly constant. Finally, the MRT demerit is constant because the workload with the smallest compact representation has a 0% MRT demerit.

7.3.3 Lessons from tradeoff study

We see from Figures 35 and 38 that running the Distiller in post-process mode can identify a much more compactly represented synthetic workload with little or no reduction in accuracy. In some cases, it is even possible to reduce the precision of an attribute and obtain a more accurate synthetic workload.

Despite the possibility of improving both accuracy and size at the same time, we see two problems with placing all attributes — even those that differ only by precision — in the library as opposed to leaving the investigation of similarly-defined attributes for the post-process step. First, some attributes (such as jump distance within state) can have more than one "axis" along which we can investigate precision

(e.g., bin width and state configuration). Varying precision along more than one axis may produce more attributes than the Distiller can evaluate in a timely manner.

Second, placing two attributes that differ only by precision in the library may lead the Distiller to choose attributes with offsetting errors instead of attributes that truly capture the target workload's performance-affecting behaviors. A synthetic workload specified using a 1 sector bin width should be more accurate than a synthetic workload specified using an 8 sector bin width. If the workload specified using an 8 sector bins size is more accurate, then the increased accuracy is the result of an offsetting error. (For example, the observed improvement in the synthetic OLTP workloads when we increased bin width from 1 to 8 was the result of an offsetting error.) Therefore, if the attribute with the 1 sector bin width does not produce an accurate synthetic workload, we want the Distiller to investigate different attributes, which may capture the performance-affecting patterns in the workload, as opposed to evaluating an attribute with an 8 sector bin width, which will be accurate only as the result of an offsetting error.

We suspect the best approach is to place only very precise attributes in the Distiller's library, then post-process the data to remove any unnecessary precision. The difficulty with this approach is that, for those attributes that divide a parameter's range into states, changing the precision also fundamentally modifies the patterns they measure.[2] For example, changing the number of states in a conditional distribution not only changes precision, but also changes the correlations between I/Os that are measured. (Recall that for jump distance within state, the state configuration that most closely matched the physical disk configuration produced the most accurate synthetic workload.) In these cases, we believe that the different state configurations need to be included directly in the library. However, we also recognize

[2] "State-based" attributes and the β-model are the only attributes in the Distiller's library for which changing precision changes the workload properties measured.

that there is much more research to be done on choosing effective state configurations for attributes containing conditional distributions.

7.4 Summary

In this chapter, we found that the demerit figures presented in Section 3.4.1 have limitations, but log area's limitations affect the distillation process less dramatically than the other demerit figures. The improvement in final RMS demerits when using log area internally instead of RMS is caused by a combination of log area's emphasis of differences at all times scales and offsetting errors. Log area works better than MRT internally because the MRT demerit figure emphasizes the workloads' randomness errors and time-dependent behaviors. The hybrid demerit figure suffers the same limitations as MRT; but, for most of our test cases, this demerit figure leads the Distiller to near-optimal final synthetic workloads. In Section 7.2, we found that raising the internal demerit threshold produced less accurate final synthetic workloads, but reduced the Distiller's running time and the size of the attributes it chose.

Finally, in Section 7.3, we saw that we can trade large, precisely-specified, accurate synthetic workloads for less precisely-specified, much smaller, but only slightly less accurate synthetic workloads. Specifically, we found that we can reduce the size of the synthetic OpenMail workload's compact representation by 70% without increasing either the log area or RMS demerits. As a result of offsetting errors, reducing the size of the synthetic OLTP workload by 5% actually doubles the accuracy. Further reductions of about 20% increase the RMS and log area demerits by about 20%.

CHAPTER 8

SYNTHETIC WORKLOAD USEFULNESS

In Chapters 6 and 7, we used the demerit figures discussed in Section 3.4.1 to evaluate the Distiller. These demerit figures allow the Distiller to run automatically in a reasonable amount of time; however, they are not necessarily the best measure of the overall quality of the attributes the Distiller chooses. Instead, we should base quality of a set of attributes (and resulting synthetic workload) on its usefulness for making design and configuration decisions.

In this chapter, we show that the Distiller specifies synthetic workloads that can predict the effects of modifying disk array configurations, even when using the moderately precise attributes evaluated in Section 7.3. Specifically, we will predict the effects of modifying the prefetch length and the RAID stripe unit size.

8.1 Prefetch length

Workloads that exhibit a high degree of spatial locality or sequentiality may benefit from *prefetching*. Upon receiving a request to read the data in sectors x_{start} through x_{end}, a disk array configured to prefetch i sectors of data will also read the data in sectors $x_{end} + 1$ through $x_{end} + i$ and place it in the cache. Prefetching improves

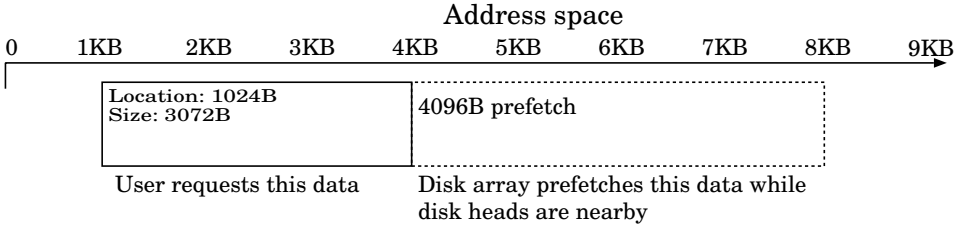

Figure 39: Example of 4KB prefetch

workload performance when the prefetched data is requested before being evicted from the cache; it degrades performance when requests for unused prefetch data delay other I/O requests. Figure 39 shows an example of prefetching when $i = 4$.

To show that the Distiller chooses attributes that contain enough information to study the effects of modifying prefetch length, we issued OpenMail (All), OLTP (All), and DSS (4LU) to a simulated disk array (described in Table 9 from Section 4.2) several times, varying the prefetch length from 0 to 2MB. We then issued the final synthetic workloads presented in Table 16 (from Section 6.2) to the simulated disk array using the same prefetch lengths and compared the mean and the 99.5th percentiles of response time.

Figures 40 through 49 show that the synthetic workloads accurately predict the general effect of prefetch length on the mean and 99.5th percentile of response time (with the exception of the DSS workload's 99.5th percentile). The remainder of this section discusses each workload separately. Specifically, we examine

1. how reducing the precision of the attributes affects their ability to predict the effects of changing prefetch length, and

2. how the observed trends relate to the values of the chosen attributes.

8.1.1 OpenMail

Figures 40 and 41 show that the original and synthetic OpenMail workloads have similar means and 99.5th percentiles of response time when issued to a disk array with a prefetch length of 64KB or less. We ran the Distiller using a disk array with no prefetching to obtain the results in Table 16 of Section 6.2. Using this original configuration, the difference between the mean response times of the original and synthetic workloads was 17%. This difference grows to only 28% as we increase prefetch length to 64KB. The difference between the 99.5th percentiles of response time grows from 8% to 38% as we increase the prefetch length from 0 to 64KB.

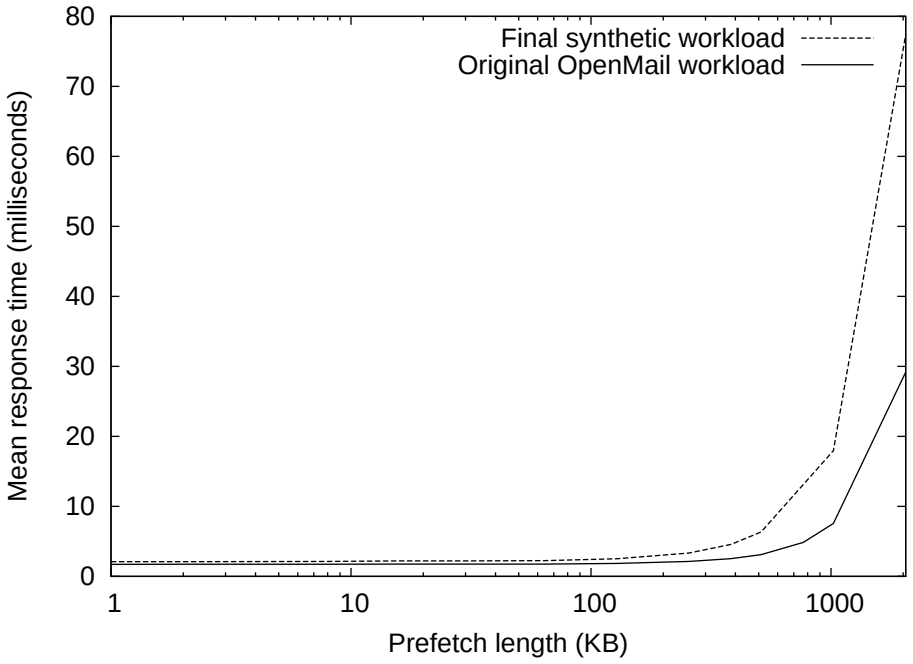

Figure 40: Mean response time of OpenMail as prefetch length varies

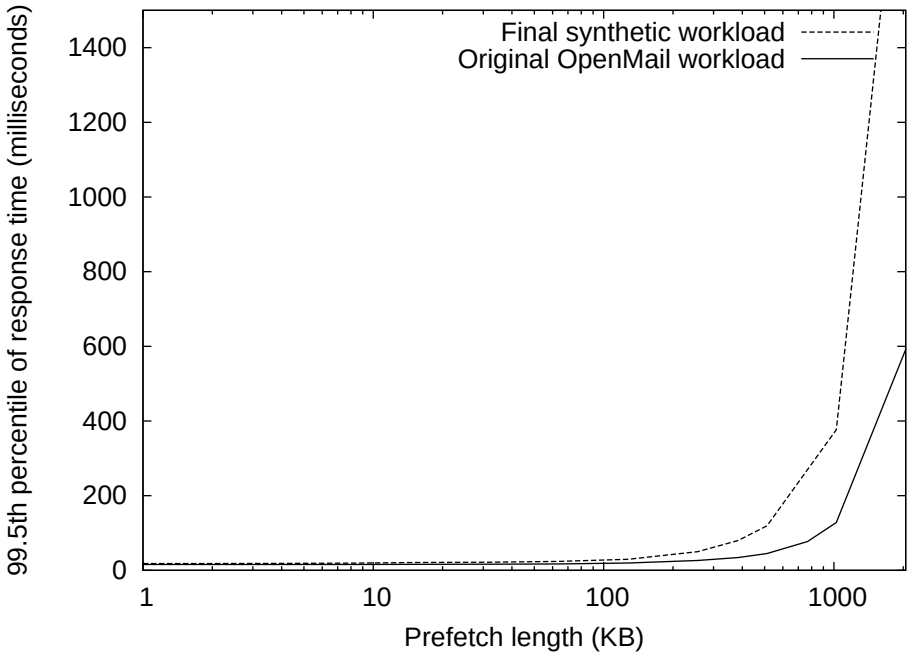

Figure 41: 99.5th percentile of OpenMail response time as prefetch length varies

We expect the difference between the mean and 99.5th percentiles of response time to grow as we increase the prefetch length because increasing the prefetch length increases the differences between the disk array under test and the disk array on which we distilled the synthetic workload. Fortunately, the synthetic workload need not predict the mean response time exactly to be useful. Knowing the general trend that increasing prefetch length increases response time enables us to choose the optimal prefetch length.

In Chapter 7.3, we saw how reducing the precision with which we measure attributes affected the external demerits of the resulting synthetic workload. We now examine how reducing the attributes' precision affects their ability to predict the effects of changing prefetch length.

Figures 42 and 43 show the effects of modifying prefetch length on several of the workloads presented in Tables 41 through 43 from Section 7.3.1. Each workload was based on some configuration of mJDWS (RW) and differs only in the bin width and number of states used. For this experiment, we compared

1. the attributes specified by the Distiller for the final synthetic workload (bin size $= 1$, $h = 2$, $s = 4$),

2. the most compact but least precise attributes tested (bin size $= 102400$, $h = 2$, $s = 4$),

3. the set of attributes with the lowest MRT demerit in Table 43 (bin size $= 8$, $h = 2$, $s = 10$), and

4. a moderately precise and moderately compact set of attributes (bin size $= 1024$, $h = 2$, $s = 4$).

In each case, the effect of modifying prefetch length is the same: Increasing prefetch length increases the mean and 99.5th percentile of response times. The main difference

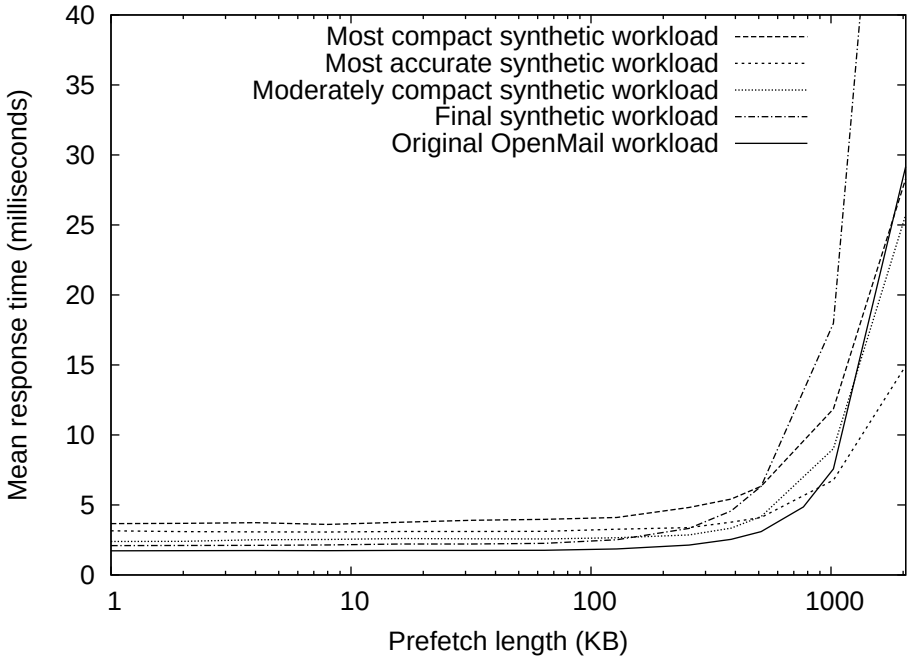

Figure 42: Effects of precision and prefetch length on OpenMail's mean response time

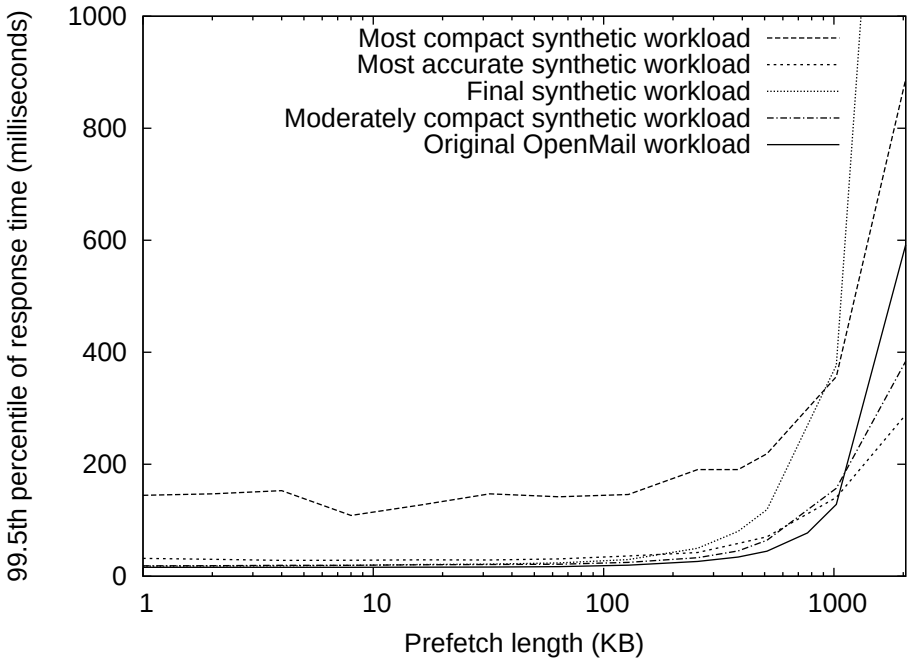

Figure 43: Effects of precision and prefetch length on OpenMail's 99.5th percentile of response time

between the workloads is how rapidly the mean and 99.5th percentile of response time increase for large prefetch lengths.

The least precise set of attributes specifies a workload whose 99.5th percentile does not strictly increase as prefetch length increases. Our investigations found no obvious cause for the fluctuations. Therefore, we assume they are simply a result of the workload's imprecise specification.

Interestingly, the moderately precise set of attributes specifies the synthetic workload that most closely tracks the increase in response times and 99.5th percentiles for large prefetch lengths. We have not yet determined why the moderately precise attributes most accurately predict the effects of modifying prefetch length; however, we suspect the accuracy is the result of offsetting errors similar to the offsetting errors in Section 7.1. To test this hypothesis, we plan to first find a set of attributes that very accurately captures OpenMail's behavior on disk arrays configured to use large prefetch lengths. We can then compare the attributes chosen for different prefetch lengths. Unfortunately, distilling workloads on disk arrays with large prefetch lengths requires attributes that are more precise than those currently present in our attribute library. In particular, we suspect we will need more precise {location} attributes.

In addition to allowing us to predict the effects of modifying prefetch length, the attribute-values themselves suggest that OpenMail (All) will not benefit from prefetching. The Distiller's first-fit algorithm chose mJDWS (RW) to describe the locality. When we examine the value of this attribute, we see that

1. more than 35% of the jumps are negative (i.e., the previous location in the same state has a larger location value),

2. more than 75% of the forward jumps are greater than 1MB, and

3. the average forward jump for each state is between 30GB and 50GB.

These figures indicate that there are very few I/Os in OpenMail (All) that potentially benefit from prefetching.

Our ability to associate the attribute-values for OpenMail (All) with its optimal prefetch length suggests a case-based approach to determining a workload's optimal prefetch length. If the number of cases is small, we can simply study many workloads and find the appropriate relationships between each workload's observed attribute-values and its optimal prefetch length. The success of such an approach will depend on the number of different cases we must study. In particular, a general solution must handle workloads for which the Distiller chooses attributes that are not related to JDWS. The similarity of the attributes chosen for our different test workloads (see Tables 17 and 18 in Section 6.2) suggests that the number of cases is manageable.

8.1.2 OLTP

The original and synthetic OLTP workloads follow the same trend as prefetch length increases. Specifically, Figures 44 and 45 show that, when the prefetch length is 128KB or less, the mean response times of the original and synthetic OLTP workloads differ by about 21%, as do the 99.5th percentiles of response time. When we raise the prefetch length above 512KB, the original workload's mean and 99.5 percentile of response time increases rapidly, whereas the synthetic workload's mean and 99.5th percentile increase only a few percent. Thus far, we have been unable to identify the cause of the disparity between the original and synthetic OLTP workloads for large prefetch lengths.

As with OpenMail (All), we explored the effects of prefetch length on several synthetic OLTP workloads specified with differing levels of precision. Each synthetic workload was based on a conditional distribution of location based on operation type and differed only in the bin width and number of states used. We explored

1. the attributes specified by the Distiller for the final synthetic workload (bin size

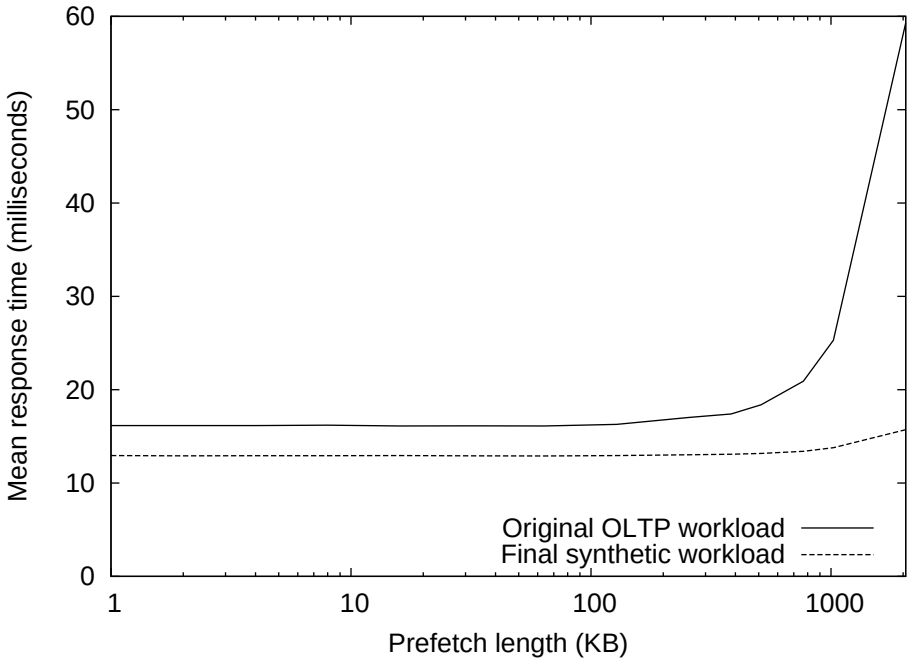

Figure 44: Mean response time of OLTP as prefetch length varies

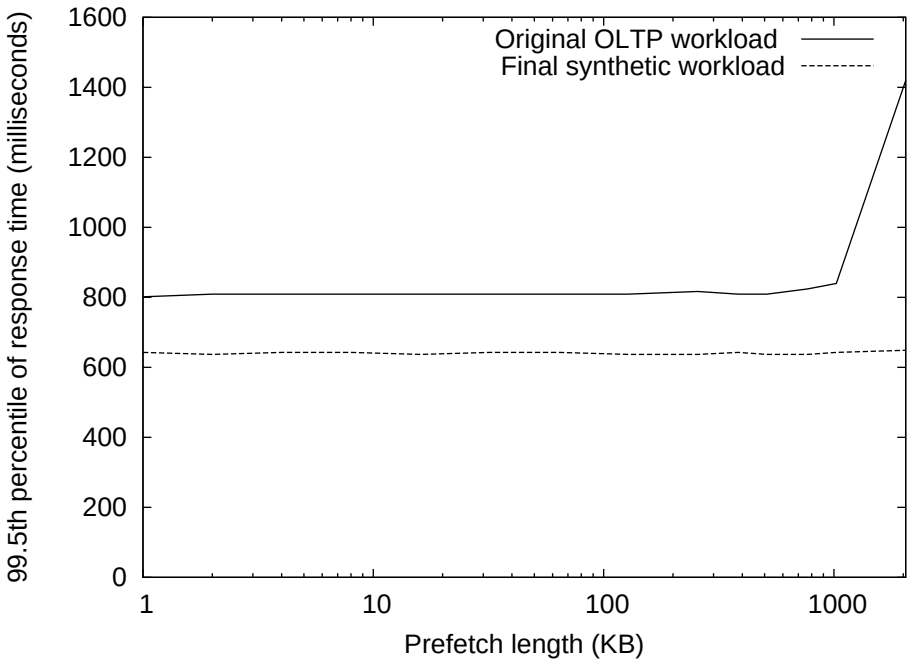

Figure 45: 99.5th percentile of OLTP response time as prefetch length varies

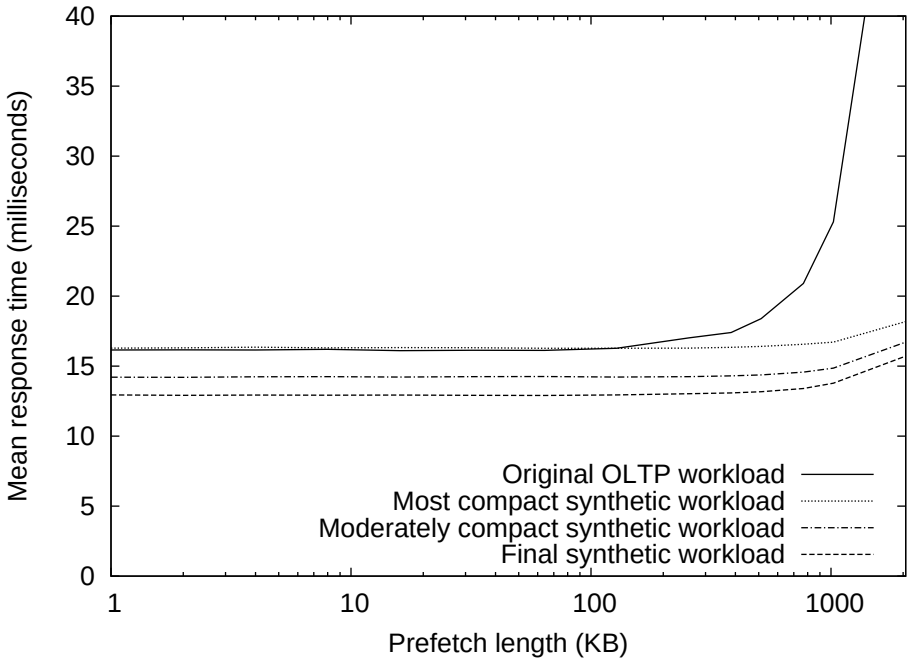

Figure 46: Effects of precision and prefetch length on OLTP's mean response time

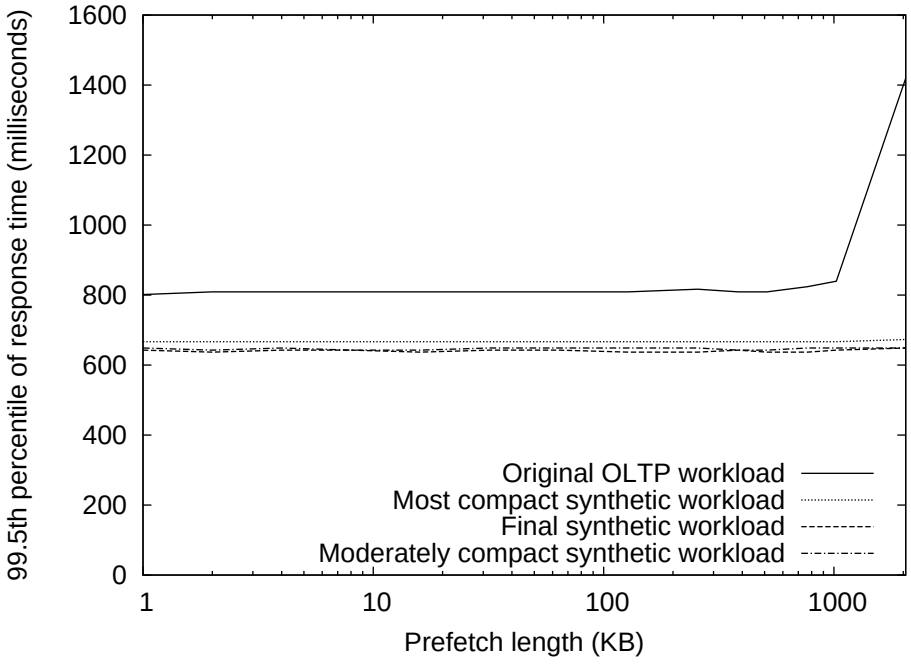

Figure 47: Effects of precision and prefetch length on OLTP's 99.5th response time percentile

$= 1$, $h = 2$, $s = 100$)[1],

2. the most compact but least precise attributes tested (bin size = 24576, $h = 1$, $s = 1$)[2],

3. a moderately precise and moderately compact set of attributes (bin size = 8, $h = 1$, $s = 1$).

Figures 46 and 47 show that the synthetic OLTP workloads demonstrate the correct trend: Prefetch length affects the mean response time only when it reaches 512KB. However, none of the synthetic workloads accurately represents the degree to which prefetch lengths over 512KB affect mean response time. (For clarity, Figures 46 and 47 present only two representative bin sizes.) This discrepancy for large prefetch lengths is understandable given that

1. we ran the Distiller with a prefetch length of 0, and

2. the final OLTP synthetic workloads are less accurate than the final OpenMail synthetic workloads.

Fortunately, as with OpenMail (All), the synthetic OLTP workloads are useful because they indicate that the optimal prefetch length is approximately 0.

Given the current library of attributes, conditional distributions of location based on operation type most accurately synthesize OLTP (All), indicating that the workload contains little or no spatial locality. This observation is consistent with Figures 44 and 45, which show that the workload does not benefit from prefetching.

8.1.3 DSS

As with OpenMail (All) and OLTP (All), Figures 48 and 49 show that the synthetic DSS workload predicts the general effects of changing prefetch length. Also like the

[1]These were also the most precise and least compact attributes.
[2]This set of attributes also produced the synthetic workload with the lowest MRT demerit.

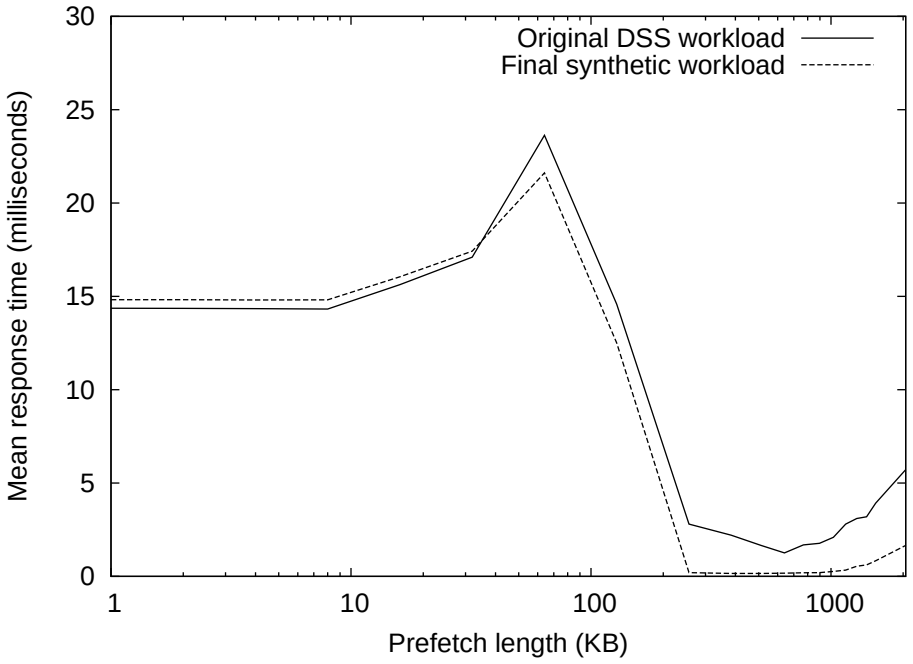

Figure 48: Mean response time of DSS as prefetch varies

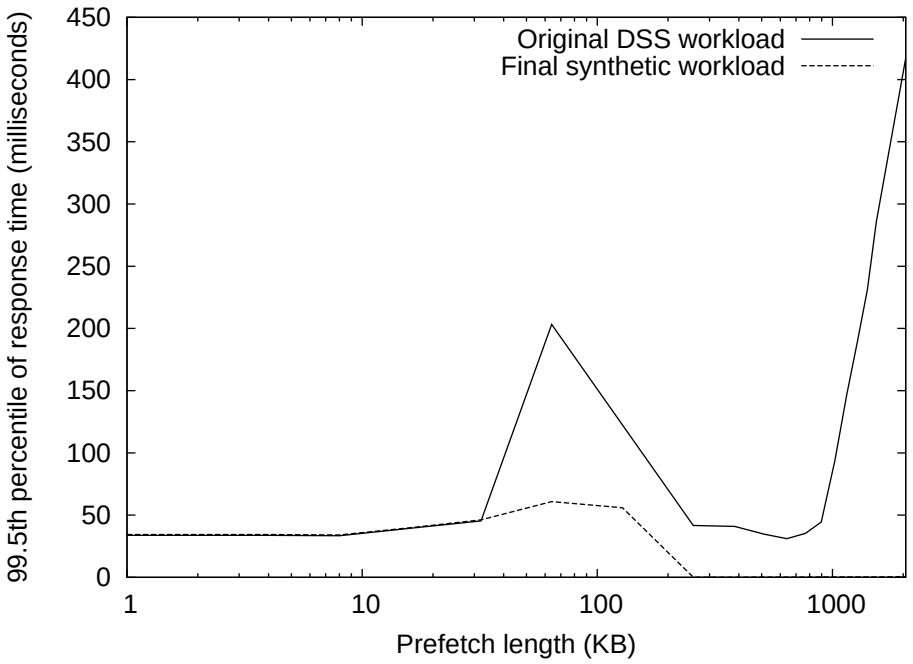

Figure 49: 99.5th percentile of DSS response time as prefetch varies

other workloads, the difference between the mean response times of the original and synthetic workloads increases as prefetch length increases (although, the differences between the mean response times are much smaller while the differences between the 99.5th percentiles are much larger).

Unlike the OpenMail and OLTP workloads, prefetching does benefit the DSS workload. The mean response time and percentile graphs have the same overall shape and indicate that the optimal prefetch length is somewhere between 256KB and 640KB. The synthetic DSS workload is so accurate, and its representation is so compact, that we did not perform any size / accuracy tradeoff experiments.

We now explain the characteristics of the DSS workload that cause the observed trends in response time. The DSS workload is a set of concurrent database queries. Each query produces several long runs of 128KB I/O requests. These interleaved runs represent a high degree of spatial locality. Consequently, we expect (as Figure 48 shows) increasing the prefetch length to improve performance. However, the causes of the increases in mean response time at 64KB and 512KB are not obvious.

The distribution of request size shows that almost every request is 128KB. When the prefetch length is less than 128KB, the prefetched data does not complete the next request. Consequently, each request requires a physical disk access. This physical access dominates the cost of servicing the request; therefore, the time spent prefetching does not significantly reduce the next request's response time. Furthermore, the prefetching delays future requests, producing an increase in mean response time as prefetch length increases from 0 to 64KB.

Mean response time also begins to increase when prefetch length reaches 512KB (4 I/Os). The cause of this trend is not clear because a prefetch length of 512KB will not fill the cache and cause it to evict unused prefetched data. One possible explanation is that "foreground" I/Os (I/Os issued by an application) become queued behind prefetch requests, thereby increasing mean response time. The simulated disk array

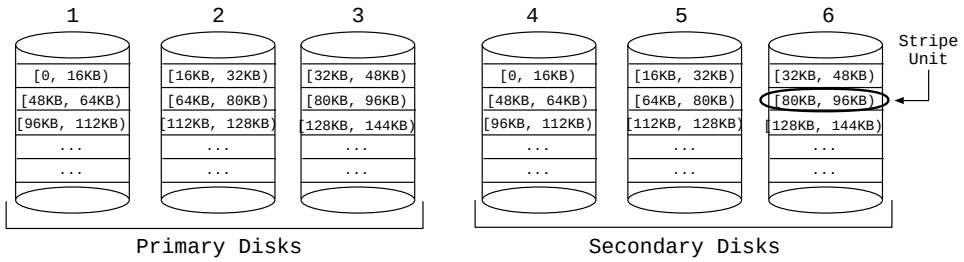

Figure 50: Example RAID 1/0 configuration with 6 disks and a 16KB stripe unit size

does not prioritize I/Os; therefore, increasing the prefetch length will increase the number of prefetch I/Os queued before foreground I/Os. This explanation, however, does not explain why the synthetic DSS workload's mean response time does not increase.

As with OpenMail, our first step toward answering this question will be improving the library so that we can distill a very accurate synthetic workload using a disk array with a large prefetch length. Because we can successfully distill an artificial version of the DSS workload with constant interarrival times, we believe that the attribute library's limitation is the {interarrival time, location} group.

8.2 Stripe unit size

Pantheon simulates disk arrays that use a RAID 1/0 (i.e., striped mirror) configuration where half of the disks in each RAID group mirror the other half. The stripe unit size defines how much data is stored contiguously on one disk before moving to the next disk in the RAID group and affects (among other things) when several disks will simultaneously service a single I/O. If a request spans several stripe units, then the requested data will be located on several disks. This parallel access can improve performance by reducing the transfer time; however, it can also degrade performance by increasing the number of seeks (i.e., causing two different disks to seek to similar locations instead of allowing the second disk to seek to the location of a different request). Figure 50 shows an example RAID 1/0 configuration with six disks and a

16KB stripe unit size. Figures 51 through 60 demonstrate the ability of the synthetic workloads to predict the effects of varying the stripe unit size from 1KB to 2MB.

8.2.1 OpenMail

Figures 51 and 52 show that the synthetic OpenMail workload demonstrates the general trend. Both the mean and 99.5th percentile of response time decrease as stripe unit size increases, until the stripe unit size reaches about 16KB. We ran the Distiller using a disk array with a 16KB stripe unit size to obtain the results in Table 16 from Section 6.2. Using this original configuration, the difference between the mean response times of the original and synthetic workloads was 17%. This difference remains steady as we increase stripe unit size, but increases to 28% as we decrease the stripe unit size. The difference between the 99.5th percentiles of response time increases from 7% to 29% as we decrease stripe unit size from 16KB to 1KB and decreases to 3% as we increase stripe unit size.

Figures 53 and 54 show that even fairly inaccurate synthetic OpenMail workloads demonstrate the correct trends. However, more precisely defined synthetic workloads more closely track the target workload's response time. The graph for the least precisely specified synthetic workload does not strictly decrease. As with prefetch length, we assume these fluctuations are simply a result of the workload's imprecise specification.

This trend of decreasing response time as stripe unit size increases is consistent with the OpenMail workload's lack of spatial locality. The stripe unit size determines how much data the disk array groups onto one disk. If a request's size is less than the size of the stripe unit, only one disk serves the request (unless the request straddles a stripe unit boundary[3]); otherwise, more than one disk serves the request. Because

[3]For OpenMail (All), when the stripe unit size is 4KB, approximately 6.5% of requests 4KB or smaller straddle a stripe unit boundary. When the stripe unit size is 8KB, approximately 2% of requests 8KB or smaller straddle a stripe unit boundary.

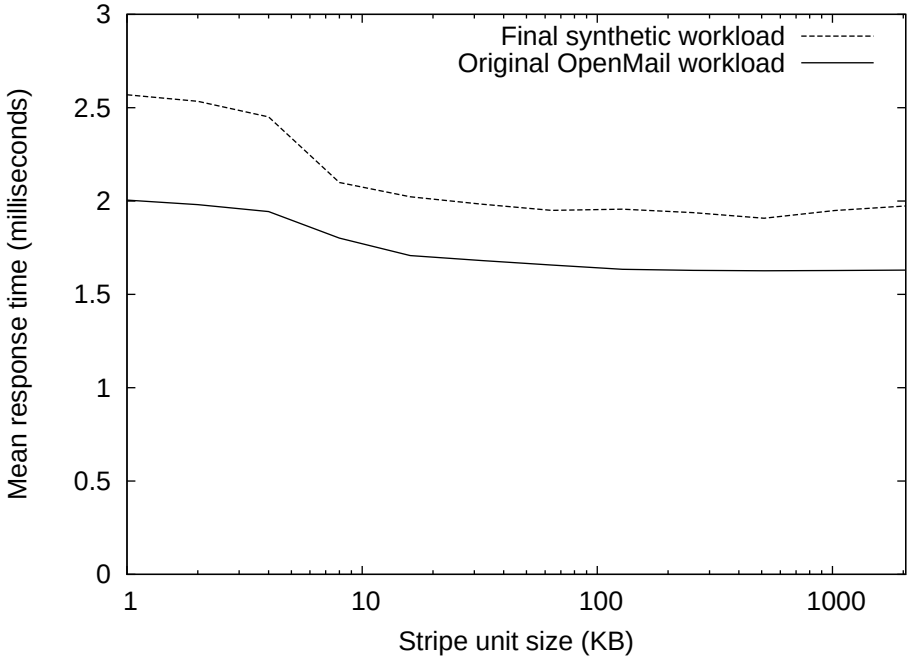

Figure 51: Mean response time of OpenMail as stripe unit size varies

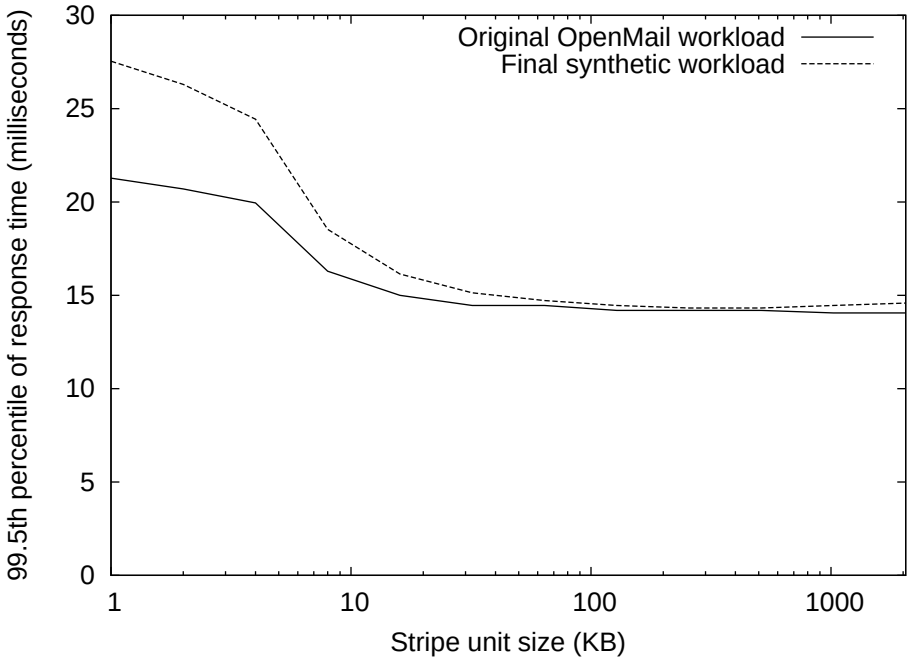

Figure 52: 99.5th percentile of OpenMail response time as stripe unit size varies

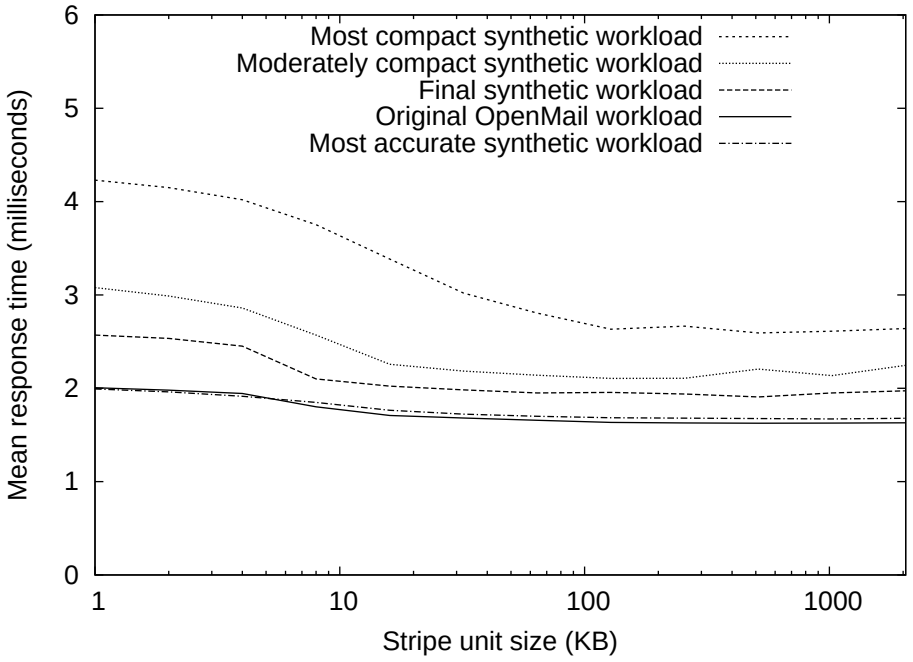

Figure 53: Effects of precision and stripe unit size on OpenMail's mean response time

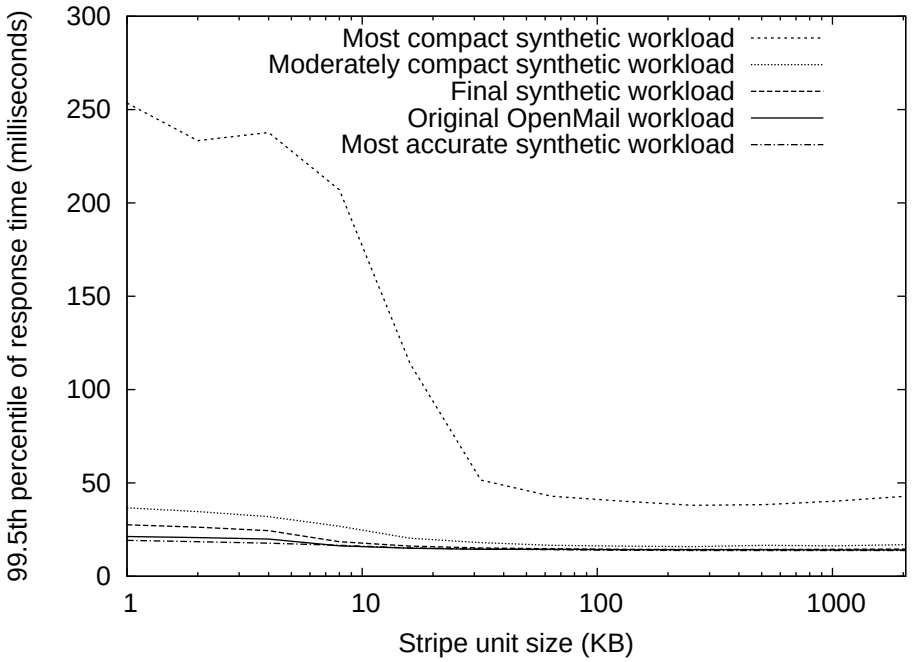

Figure 54: Effects of precision and stripe unit size on OpenMail's 99.5th percentile of response time

the OpenMail workload has little spatial locality, we expect seek time to dominate the cost of serving requests. Using a large stripe unit size will cause all requests to be served by only one disk, leaving the second disk free to serve future requests. If more than one disk serves each request, then future requests will be queued while waiting for all disks to move their heads.

Notice also that the mean response time decreases rapidly at first, then levels off for larger stripe unit sizes. A large percentage of the workload's requests are 1KB and 8KB. Thus, increases in stripe unit size beyond 8KB have little effect on mean response time.

8.2.2 OLTP

When distilling OLTP (All), we simulated a disk array similar to the one on which the OLTP workload trace was collected. This simulated disk array has two physical disks per LU; therefore, stripe unit size has no effect on data layout and no effect on performance. To present a more interesting result, we re-ran the experiment simulating a disk array with four disks per LU. We used four-disk LUs to generate all results regarding stripe unit size for OLTP (All).

Figures 55 and 56 show behavior similar to the OpenMail stripe unit size test. However, in this case, the performance levels off after the stripe unit size reaches 2KB. This is consistent with the fact that over 90% of the OLTP workload's requests are 2KB. (In addition, fewer than 1% small I/Os straddle stripe unit boundaries.)

Figures 57 and 58 show that the user can trade a considerable amount of accuracy for compact representation size and still have a synthetic workload that exhibits the correct trend as stripe unit size increases. However, only precisely defined synthetic workloads exhibit the sharp drop in mean response time when the stripe unit size increases from 1KB to 2KB. In contrast, less precisely defined synthetic workloads exhibit a more gradual decrease in response time as stripe unit size increases.

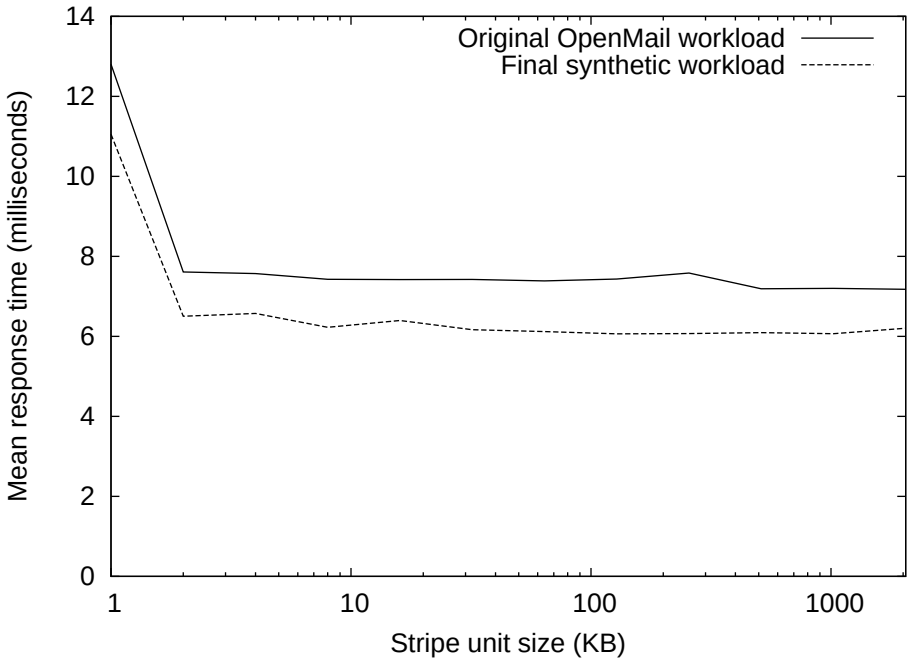

Figure 55: Mean response time of OLTP as stripe unit size varies

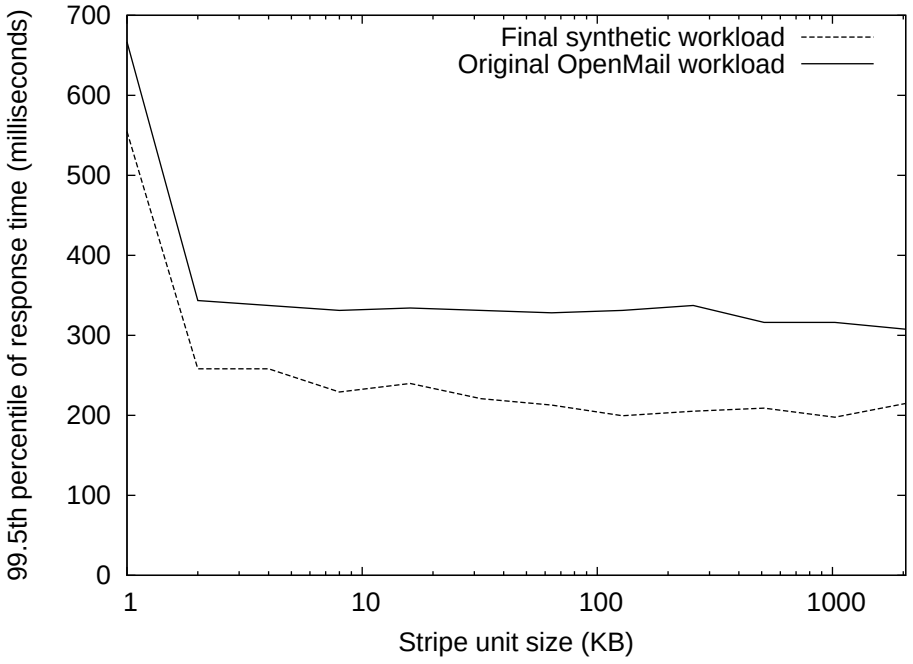

Figure 56: 99.5th percentile of OLTP response time as stripe unit size varies

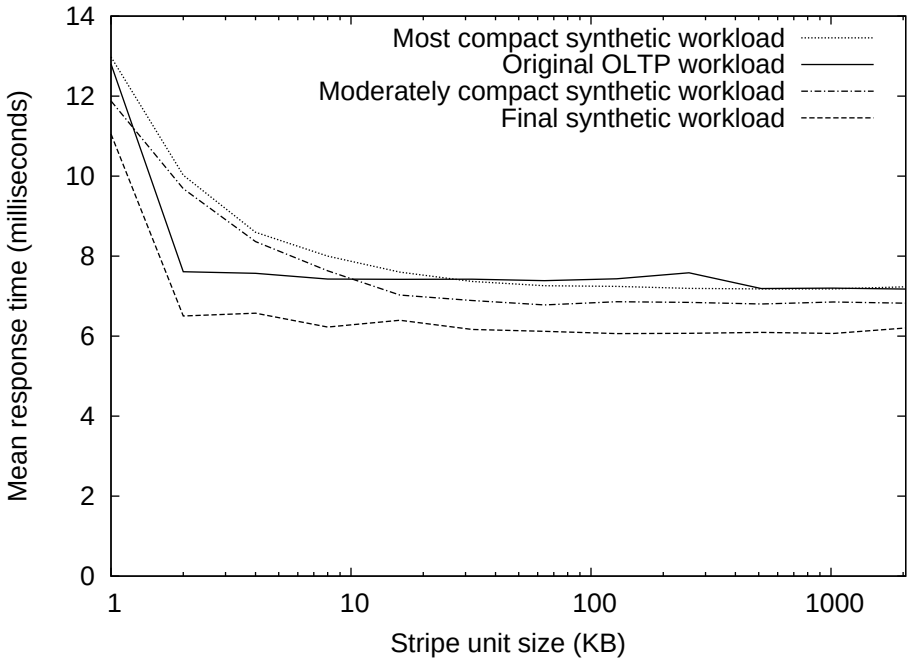

Figure 57: Effects of precision and stripe unit size on OLTP's mean response time

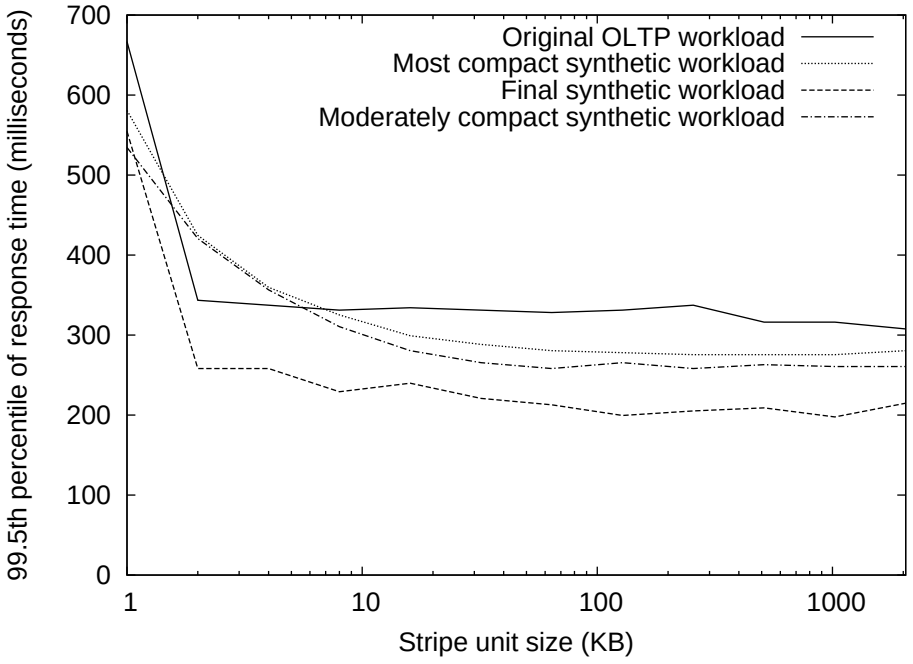

Figure 58: Effects of precision and stripe unit size on OLTP's 99.5th percentile of response time

8.2.3 DSS

As with prefetch length, we see in Figures 59 and 60 that the mean response times of the target and synthetic DSS workloads remain close at all stripe unit sizes. As before, there are no size / accuracy tradeoffs to examine for the DSS workload.

The observed changes in performance as stripe unit size increases make sense given Pantheon's RAID 1/0 data layout algorithm. Suppose an LU contains exactly four interleaved, synchronized streams. When the stripe unit size is less than 128KB, each request covers at least two stripe units; therefore, two disks serve each request. Furthermore, because there are multiple interleaved streams and zero prefetching, these two disks both suffer a seek for each I/O serviced. (The disk array alternates between the primary and mirror disks when reading data. It also bundles contiguous data from one I/O request into a single physical disk request. Therefore, two disks serve each requests when the stripe unit size is less than 128KB.)

When the stripe unit size is 128KB, each request covers exactly one stripe unit. (Only 2% of the DSS workload's I/Os straddle 128KB stripe unit boundaries.) Therefore, only one disk serves each request. The mean request size increases because, as with smaller stripe unit sizes, each request suffers a seek, but there is no second disk to reduce the transfer time.

When the stripe unit size is 256KB or greater, it becomes possible for some I/Os not to suffer a seek. If two of the streams begin with accesses to stripe units on disks 1 and 3, and the other two streams begin with accesses to stripe units on disks 2 and 4, then the same disk will serve every pair of requests on a given stream. Figure 61 shows which disks serve which requests for a given stripe unit size.

8.3 Summary

A synthetic workload's true quality is the degree to which it can be used place of the target workload without affecting results of a given evaluation. In this chapter,

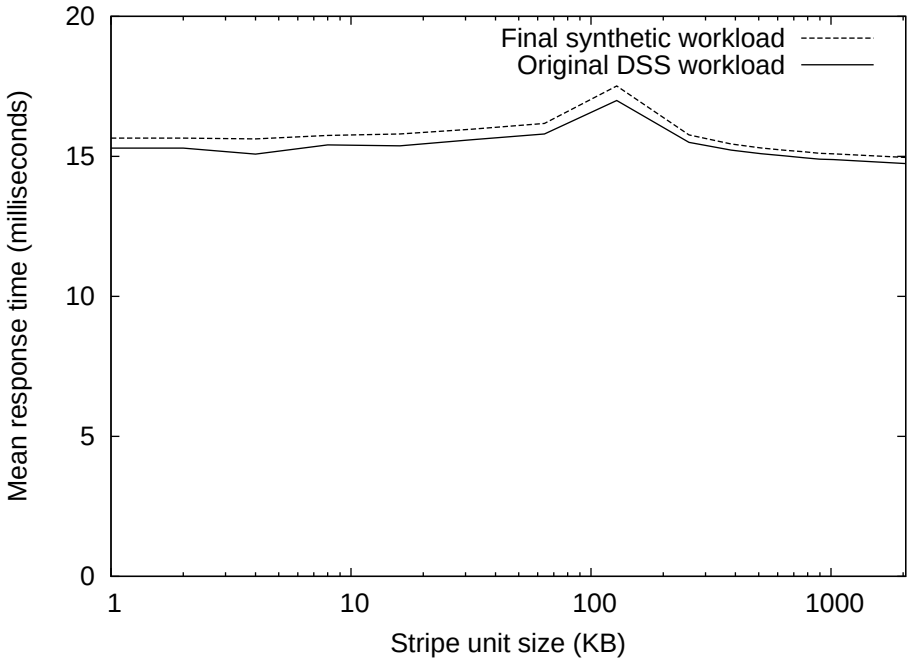

Figure 59: Mean response time of DSS as stripe unit size varies

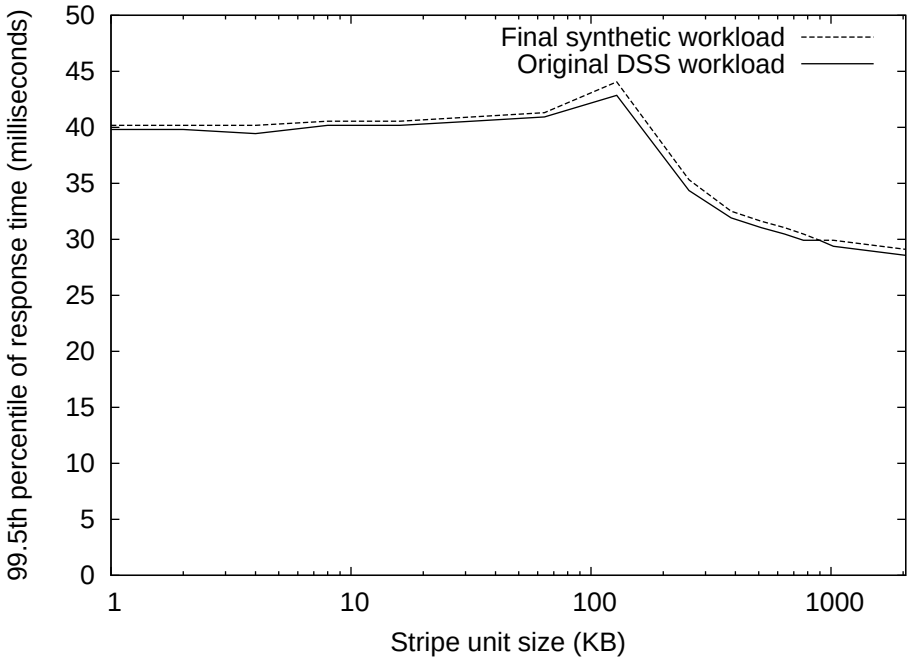

Figure 60: 99.5th percentile of DSS response time as stripe unit size varies

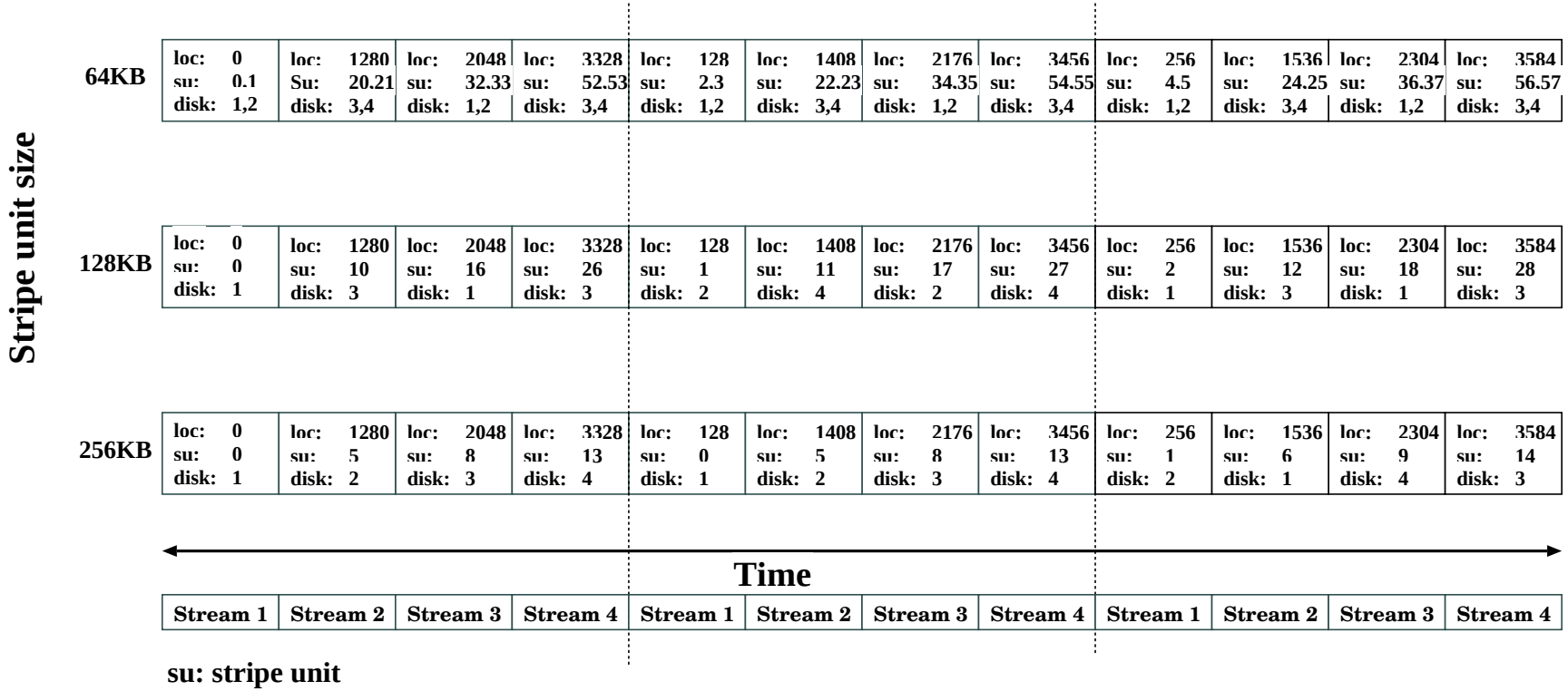

Figure 61: Effect of stripe unit size on DSS disks accessed

we demonstrated that evaluators can use the synthetic workloads specified by the Distiller in place of target workloads when evaluating the effects of modifying a disk array's prefetch length and stripe unit size. They predict the optimal prefetch length for OpenMail (All) and OLTP (All), and predict the optimal stripe unit size for OpenMail (All), OLTP (All), and DSS (4LU). Furthermore, the less precisely specified synthetic workloads studied in Chapter 7.3 also predict the optimal prefetch length and stripe unit size.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

This dissertation presented the design and evaluation of the Distiller, our tool for automatically identifying which attribute-values two workloads must share in order to have similar performance on a given disk array. These attribute-values can serve as a description of a set of representative synthetic workloads, thereby increasing the limited set of available evaluation workloads. More importantly, the chosen attribute-values help highlight the important performance-affecting interactions between the disk array and the workload, thereby aiding the development of disk arrays and self-configuring storage systems.

The Distiller automatically chooses attributes using a novel, iterative approach: Beginning with a trace of a production workload and a library of candidate attributes partitioned into groups attribute groups, it quickly estimates whether a group contains any key performance-affecting attributes, then searches those groups that potentially contain a key attribute.

In Chapter 6, we demonstrated that the Distiller finds synthetic workloads with log area demerits less than 10% for six out of the eight workloads we tested. The cause of the high demerit in the remaining two cases appears to be primarily a limitation of the attribute library. In Chapter 7 we found that all of the potential demerit figures used to compare workload performance had limitations. The Distiller produced the most accurate synthetic workloads when using the hybrid demerit figure for its internal evaluations; however, the distillation process was least affected by the Distiller's limitations when using the log area demerit figure. We also found that the internal threshold used to determine which attribute groups to investigate and which

attributes to add to the list of key attributes had little effect on the accuracy of the final synthetic workloads. However, in all cases, the threshold did have a large effect on the Distiller's running time and the size of the final synthetic workloads' compact representations. Section 7.3 found that, for OpenMail and OLTP, reducing the precision with which we measured attributes resulted in considerably smaller compact representations, but only moderately less accurate synthetic workloads. Finally, Chapter 8 shows that we can use the information contained in the chosen attributes to predict the performance effects of modifying the storage system's prefetch length and stripe unit size. In addition, we discussed how the values of the chosen attributes highlight the causes of the observed trends.

In summary, we found that the Distiller was useful for both finding the description of representative synthetic workloads, and for helping to highlight the interactions between a workload and storage system.

In the future, we plan to address the limitations of the Distiller that our evaluations highlighted. First, we plan to investigate techniques for reducing the effects of random error, particularly when using the mean response time demerit figure. Second, we want to better understand the effects of the rotation amount on the Distiller's choice of attribute groups and attributes and reduce the dependence of the Distiller on those effects. After addressing these limitations, we will re-evaluate our demerit figures. Finally, we would like to evaluate how our use of an open workload model instead of a closed workload model affects our results — especially when distilling the DSS workload.

In the longer term, we want to improve the Distiller in several areas:

1. **Attribute library:** While evaluating the Distiller, we saw several examples of where the accuracy of the final synthetic workload appeared to be the result of offsetting errors. Offsetting errors occur when the attribute library does not contain an attribute that meets the specified threshold and the Distiller chooses

the best available attribute. Adding attributes that meet the threshold will help reduce the incidence of offsetting errors.

2. **Library of target workload traces:** Increasing the number of target workloads will help provide a broader perspective and add confidence to our results.

3. **Automatic inclusion/exclusion of attributes:** We believe that some attributes can be ruled out as key attributes by simply examining their values. For example, a run count attribute for which all the run counts are 1 is not a key attribute because the default distribution of location values will produce a workload with the same attribute-value.

4. **Multiple trials for confidence:** As our available resources grow, we will begin automatically generating and evaluating multiple copies of each workload, thereby providing improved statistical confidence and potentially reducing the effects of randomness error on the MRT demerit figure.

The cost of configuring and managing storage systems is quickly overtaking the cost of purchasing the hardware. Therefore, we are looking forward to contributing to the research of self-managing storage by continuing our study of how attributes highlight the interactions between the workload and storage system. In Chapter 8, we saw how the chosen attribute-values were consistent with the observed performance trends given the changes in prefetch length and stripe unit size. We would like to continue to study those interactions and produce a model that predicts optimal prefetch length and stripe unit size given the chosen attribute-values. Such a model will be instrumental in developing self-managing disk arrays.

Finally, we believe the Distiller's techniques generalize to other areas of computer systems design. We look forward to investigating how the we can use the Distiller to identify the key attributes of other workloads and possibly use those attributes a basis for other autonomic systems.

APPENDIX A

GENERATION TECHNIQUES

The Distiller operates independently of the tool used to generate synthetic workloads. It requires only that the generation tool be able to produce a synthetic workload with a specified set of attribute-values. However, because synthetic workload generation is often a non-trivial problem, this section discusses our generation techniques.

Our synthetic workload generation tool, *GenerateSRT*, coordinates the operation of several individual *generators*. We implement one generator for each attribute in the Distiller's library. Each generator is responsible for producing the values for the request parameters measured by the attribute such that the synthetic workload exhibits the measured attribute-value. Two generators that produce values for the same I/O request parameter will interfere with each other; therefore, a workload may be specified using at most one attribute from each attribute group.

Allowing only one attribute per attribute group is not a limitation of the generation tool; instead, it is a consequence of the need for special algorithms to produce two or more attributes simultaneously. For example the algorithm that produces both the desired distribution of location and the desired distribution of jump distance is considerably more complicated than the algorithms that produce these distributions individually. We make this new algorithm available to the Distiller by defining a new attribute that measures and generates both the distribution of location and the distribution of jump distance. We then incorporate that algorithm into a single generator.

Some generators corresponding to attributes in two-parameter attribute groups produce values for only one of the two parameters. For example, a generator that produces separate distributions of location for read requests and write requests (e.g.,

CD(op. type, location, 1, 2)) generates only location values. We can use any {operation type} generator to produce the operation types, then use this {operation type, location} generator to produce locations based on the current operation type. Notice, however, that not every {operation type, location} attribute is compatible with (i.e., will not interfere with) every {location} attribute. For example, most {location} attributes will be incompatible with a joint distribution of operation type and location. The description of each attribute in the Distiller's library includes a list of the corresponding generator's dependencies and restrictions. The Distiller is able to use this information to selects sets of attributes that will not interfere.

The remainder of this section discusses several our non-trivial generation techniques.

A.1 Distributions

The generator corresponding to the conditional distribution attribute discussed in Section 3.3 is very straightforward: It draws the value(s) for the dependent parameter(s) from the conditional distribution specified by the value(s) of the independent parameter(s). For example, the generator corresponding to CD(operation type, location, 1, 2) draws the value for the current I/O's location from either the distribution of read locations or the distribution of write locations based on the operation type chosen for the current I/O. Because this generator chooses location based on operation type, GenerateSRT must be configured to run the generator producing operation types before CD(operation type, location, 1, 2).

When using several conditional distribution generators simultaneously, one must resolve any dependences and avoid "loops". Circular dependencies such as CD(operation type location, 1, 2) and CD(location, operation type, 1, 100) will cause the workload generator to fail. The Distiller automatically identifies all dependencies and terminates if there is a circular dependency.

Table 48: Sample transition matrix for operation type

	read	write
read	.75	.25
write	.20	.80

A.2 Transition matrices

When given a transition matrix, the Distiller chooses the next state from the probability distribution given by the row corresponding to the current state. For example, given the transition matrix shown in Table 48, and given that the current state is "write", then the Distiller will chose the next state to be "read" with probability .2 and "write" with probability .8.

The generator corresponding to STC adds an additional step: Once it chooses a state, it then draws a run count x from the run count distribution corresponding to that state. The generator then remains in the chosen state for x I/Os before choosing another state and run count.

A.3 Jump distance

The simplest technique for reproducing a distribution of jump distances is to choose an initial location, then choose successive locations based on a randomly chosen jump distance. There are two problems with using this technique: First, the result of the randomly chosen jump may be an invalid location (e.g., a location that is out of range for the storage device). Ignoring the random draws that lead to invalid locations will skew the resulting distribution. Second, the resulting distribution of location values is unlikely to match that of the target workload. (The distribution of location values need not be part of the attribute; however, in practice, we believe that any useful jump distance attribute will also maintain the distribution of location.)

Our solution is to randomly draw the set of location values and the set of jump distances from the specified distributions, then find an ordering of the locations and

jump distances such that

$$location_i + jump\_distance_i = location_{i+1}$$

We suspect that an exact solution to this problem is NP-Complete (it appears to reduce to the Hamiltonian Path problem). Furthermore, because the set of locations and jump distances are drawn randomly from a distribution, it is possible that there is no exact solution. Consequently, we generate an approximate solution using the greedy algorithm in Figure 62:

This algorithm will maintain the desired distribution of location values (because it chooses only location values on the list); however, it is not guaranteed to maintain the distribution of jump distance. If it does not find any valid jump distances for a particular location, it randomly chooses a location from the location distribution (it does not backtrack). In addition, we specify a threshold within which a chosen jump distance may vary. For example, if the current location is 16, the chosen jump distance is 60, and the threshold is 4, we allow the generator to choose any location from the distribution between 72 and 80. Consequently, in pathological cases, the jump distance distribution could differ significantly from the desired distribution. Fortunately, we have found that, in practice, the resulting distribution is close enough to produce a representative synthetic workload.

- **Jump distance within state:** To generate a workload with the specified "jump distance within state" attribute-value, we separately generate the sequence of locations for each state (using the jump distance generation technique shown in Figure 62). We then interleave these separate streams of locations according to the state transition matrix.

- **Jump distance within state (RW):** The JDWS (RW) generator is nearly identical to the JDWS generator. The only difference is that the states in the

```
// location is a hash table of locations chosen randomly from a
// given distribution.

// jump_distance is a linked lists containing the
// set of jump distances to be used.

let jd_index := head_of_list_ptr(jump_distance);
let current_location  := get_random_location(location);

while (! list_empty(location))
{

    let starting_index := jd_index;
    let proposed_location :=
         nearest_location(current_location +
                          get_value(jump_distance, jd_index));
    while (abs(proposed_location - current_location +
                 get_value(jump_distance, jd_index) < threshold))
    {
       jd_index := get_next_index(jump_distance, jd_index);
       if ( jd_index = starting_index )
       {
          proposed_location = get_random_location(location);
          break;
       }
    }

    set_location(proposed_location)
    remove_from_list(jump_distance, jd_index);
    remove_from_table(proposed_location);
    let current_location = proposed_location +
 get_current_request_size();
}
```

Figure 62: Greedy algorithm for choosing jump distances

transition matrix specify both the location state and the operation type for the current I/O.

Because states are (operation type, location) pairs, the state transition matrix implicitly defines a Markov model of operation type. Should the user choose not to generate operation type using the implicit Markov model, she can set the *pol* ("preserve operation list") flag. The synthetic workload generator will then use a sequence of operation types described by a different attribute (e.g., operation type run count).

- **Modified jump distance:** The generators for modified jump distance, modified jump distance within state, and modified jump distance within state (RW) are nearly identical those for jump distance, distance within state, and jump distance within state (RW). The only difference is that code in Figure 62 does not add the value of the current request size when determining the current location.

A.4 Run count

To generate a workload with the specified run count, the Distiller chooses a location for the head of the run using any user-specified generation technique (distribution, conditional distribution, jump distance within state, etc.), draws a run length x from the distribution of run lengths, then generates the next $x - 1$ location values such that

$$location_i = location_{i-1} + request\_size_{i-1}$$

- **Run count within state:** To generate a workload with the specified "run count within state" attribute-value, we separately generate the sequence of locations for each state. We then interleave these separate streams of locations

according to the state transition matrix.

- **Run count within state (RW):** The run count within state (RW) generator is nearly identical to the run count within state generator. The only difference is that the states in the transition matrix specify both the location state and the operation type for the current I/O.

- **Modified run count:** The modified run count generator does not consider the current I/O's request size when generating a run (nor do the modified run count within state, and modified run count within state (RW) generators). Therefore, the distance between I/Os in a run is either (1) a constant specified by the user, or (2) drawn randomly from the distribution of modified jump distances.

- **Operation type run count:** This generator alternates between runs of read requests and write requests. It draws the length of each run from the appropriate distribution of run counts for reads or writes.

- **Interleaved runs:** For each interleaved run, the interleaved runs attribute specifies

  1. beginning location,
  2. number of I/Os,
  3. starting I/O number (relative to beginning of trace), and
  4. ending I/O number

  The generator distributes an interleaved run's I/Os uniformly over the appropriate range of the trace. For example, if an interleaved run has 10 I/Os with a starting I/O of 50 and ending I/O of 100, then approximately every fifth I/O between 50 and 100 will be from this interleaved run. The uniform spacing is rounded to the nearest integer; and ties are broken arbitrarily.

A.5 Burstiness

β-model: For each window, this attribute presents the number of I/Os in the window m, and a parameter β. The generator takes βm I/Os and allocates them to one randomly chosen half of the window (first half or second half). It allocates the remaining $(1-\beta)m$ I/O to the other half of the window. It then repeats this process recursively on each half of the window until either (1) each sub-window has at most 1 I/O, or (2) the length of the sub-window falls below the user-specified sensitivity parameter. In the second case, the generator spreads the remaining I/Os uniformly across the sub-window. See [56] for full details.

IAT clustering: The IAT clustering attribute divides the trace into intervals and selects a representative arrival pattern for each interval. The generation technique simply concatenates copies of the chosen representative segments into a single arrival pattern. See [33] for details.

APPENDIX B

DEMERIT FIGURE ALGORITHMS

This section contains the pseudo-code for the RMS and log area metrics. Each metric compares two cumulative distribution functions (CDFs) of response time. A CDF of response time is an increasing function $f : \mathbb{R} \rightarrow \mathbb{R}$ where $f(x)$ is defined to be the fraction of I/Os whose response time is at most x.

B.1 Root-mean-square

Given two CDFs of response time f_1 and f_2, the root-mean-square difference between f_1 and f_2 is calculated as follows:

```
sum := 0
for y = 0.0 to 1.0 step .001
begin
    difference := f_1^{-1}(y) − f_2^{-1}(y)
    sum += difference^2
end
 return square_root(sum)
```

Our RMS demerit figure is the above RMS metric divided by the mean value for f_1.

B.2 Log area

Given two CDFs of response time f_1 and f_2, the log area is calculated as follows:

sum := 0

for y = 0.0 to 1.0 step .001 begin

    if ($f_1^{-1}(y) > f_2^{-1}(y)$) begin

        difference := $\frac{f_1^{-1}(y)}{f_2^{-1}(y)}$

    end

    else begin

        difference := $\frac{f_2^{-1}(y)}{f_1^{-1}(y)}$

    end

sum += difference^2

end

return square_root(sum)

We calculate our log area demerit figure by subtracting the above log area metric from 1.0.

REFERENCES

[1] Alvarez, G. A., Borowsky, E., Go, S., Romer, T. H., Becker-Szendy, R., Golding, R., Merchant, A., Spasojevic, M., Veitch, A., and Wilkes, J., "Minerva: an automated resource provisioning tool for large-scale storage systems," *ACM Transactions on Computer Systems*, November 2001.

[2] Anderson, E., Hobbs, M., Keeton, K., Spence, S., Uysal, M., and Veitch, A., "Hippodrome: Running circles around storage administration," in *Proceedings of the Conference on File and Storage Technologies*, pp. 175–188, IEEE, January 2002.

[3] Anderson, E., Kallahalla, M., Spence, S., Swaminathan, R., and Wang, Q., "Ergastulum: an approach to solving the workload and device configuration problem," tech. rep., Hewlett-Packard Laboratories, http://www.hpl.hp.com/SSP/papers/, 2001.

[4] Anderson, E., Swaminathan, R., Veitch, A., Alvarez, G. A., and Wilkes, J., "Selecting RAID levels for disk arrays," in *Proceedings of the Conference on File and Storage Technologies*, January 2002.

[5] Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., Leung, S.-T. A., Sites, R. L., Vandevoorde, M. T., Waldspurger, C. A., and Weihl, W. E., "Continuous profiling: Where have all the cycles gone?," *ACM Transactions on Computer Science*, vol. 15, pp. 357–390, November 1997.

[6] Baker, M. G., Hartman, J. H., Kupfer, M. D., Shirriff, K. W., and Ousterhout, J. K., "Measurements of a distributed file system," in *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pp. 198–212, ACM, 1991.

[7] Berry, M. R. and El-Ghazawi, T. A., "An experimental study of input/output characteristics of NASA earth and space sciences applications," in *Proceedings of IPPS '96, The 10th International Parallel Processing Symposium*, pp. 741–747, IEEE, 1996.

[8] Blum, A. L. and Langley, P., "Selection of relevant features and examples in machine learning," *Artifical Intelligence*, no. 1-2, pp. 245–271, 1997.

[9] Bodnarchuk, R. R. and Bunt, R. B., "A synthetic workload model for a distributed system file server," in *Proceedings of SIGMETRICS*, pp. 50–59, 1991.

[10] Boyd, W. T. and Recio, R. J., *Workload Characterization: Methodology and Case Studies, 1999*, ch. "I/O workload characteristics of modern server", pp. 87–96. 1999.

[11] Bunt, R. B., Murphy, J. M., and Majumdar, S., "A measure of program locality and its application," in *Proceedings of SIGMETRICS*, pp. 28–40, ACM, 1984.

[12] Calingaert, P., "System performance and evaluation: survey and appraisal," *Communications of the ACM*, vol. 10, pp. 12–18, Jan 1967.

[13] Calzarossa, M. and Serazzi, G., "Workload characterization: A survey," *Proceedings of the IEEE*, vol. 81, pp. 1136–1150, August 1993.

[14] Calzarossa, M. and Serazzi, G., "Construction and use of multiclass workload models," *Performance Evaluation*, vol. 19, pp. 341–352, 1994.

[15] Conte, T. M. and Hwu, W. W., "Benchmark characterization for experimental system evaluation," in *Proceedings of the 1990 Hawaii International Conference on System Sciences*, vol. I, pp. 6–18, 1990.

[16] Ebling, M. R. and Satyanarayanan, M., "SynRGen: an extensible file reference generator," in *Proceedings of SIGMETRICS*, pp. 108–117, ACM, 1994.

[17] Ferrari, D., *Computer Systems Performance Evaluation*. Prentice-Hall, Inc., 1978.

[18] Ferrari, D., "Characterization and reproduction of the referencing dynamics of programs," *Proceedings of the 8th International symposium on computer performance, Modeling, Measurement, and Evaluation*, 1981.

[19] Ferrari, D., "On the foundations of artificial workload design," in *Proceedings of SIGMETRICS*, pp. 8–14, 1984.

[20] Ferrari, D., Serazzi, G., and Zeigner, A., *Measurement and Tuning of Computer Systems*. Prentice-Hall, Inc., 1983.

[21] Ganger, G. R., "Generating representative synthetic workloads: An unsolved problem," in *Proceedings of the Computer Measurement Group Conference*, pp. 1263–1269, December 1995.

[22] Gill, D. S., Zhou, S., , and Sandhu, H. S., "A case study of file system workload in a large-scale distributed environment," in *Proceedings of SIGMETRICS*, pp. 276–277, ACM, 1994.

[23] Golding, R., Staelin, C., Sullivan, T., and Wilkes, J., "'Tcl cures 98.3% of all known simulation configuration problems' claims astonished researcher!," in *Proc. of Tcl Workshop*, May 1994.

[24] Gomez, M. E. and Santonja, V., "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 Telecommunication Systems*, pp. 199–206, IEEE, 2000.

[25] Gomez, M. E. and Santonja, V., "A new approach in the analysis and modeling of disk access patterns," in *Performance Analysis of Systems and Software (ISPASS 2000)*, pp. 172–177, IEEE, April 2000.

[26] Gribble, S. D., Manku, G. S., Roselli, D., Brewer, E. A., Gibson, T. J., and Miller, E. L., "Self-similarity in file systems," in *Proceedings of SIGMETRICS*, pp. 141–150, 1998.

[27] Hewlett-Packard Corporation, "Lintel." http://www.hpl.hp.com/research/ssp/software/index.html.

[28] Hewlett-Packard Corporation, "Model 30/FC high availability disk array - user's guide." Pub No. A3661-90001, August 1998.

[29] Hewlett-Packard Corporation, December 2000.

[30] Hewlett-Packard Corporation and HP OpenView Integration Lab, *HP OpenView Data Extraction and Reporting*. Hewlett-Packard Company, Available from http://managementsoftware.hp.com/library/papers/index.asp, version 1.02 ed., February 1999.

[31] Hilegas, J. R., Nester, A. C., Gosden, J. A., and Sisson, R. L., "Computer design: Generalized measures of computer system performance," in *Proc. of the 1962 ACM national conference on Digest of technical papers*, sep 1962.

[32] Hong, B. and Madhyastha, T., "The relevance of long-range dependence in disk traffic and implications for trace synthesis," tech. rep., University of California at Santa Cruz, 2002.

[33] Hong, B., Madhyastha, T., and Zhang, B., "Cluster-based input/output trace synthesis," tech. rep., University of California at Santa Cruz, 2002.

[34] Joslin, E. O., "Applications benchmarks: the key to meaningful computer evaluations," in *Proc. of the ACM 20th National Conference*, pp. 27–37, 1965.

[35] Kao, W. and Iyer, R. K., "A user-oriented synthetic workload generator," in *Proceedings of the 12th International Conference on Distributed Computing Systems*, pp. 270–277, 1992.

[36] Keeton, K., Veitch, A., Obal, D., and Wilkes, J., "I/O characterization of commercial workloads," in *Proceedings of 3rd Workshop on Computer Architecture Support using Commercial Workloads (CAECW-01)*, January 2001.

[37] Laha, S., Patel, J., and Iyer, R., "Accurate low-cost methods for performance evaluation of cache memory systems," *IEEE Transactions on Computers*, vol. 37, pp. 1325–1336, November 1998.

[38] Leutenegger, S. T. and Dias, D., "A modeling study of the TPC-C benchmark," in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pp. 22–31, ACM Press, 1993.

[39] Mahgoub, I., "Storage subsystem workload characterization in a real life personal computer environment," in *Conference Record of Southcon 95*, pp. 212–216, IEEE, March 1995. ISBN = 0-78032576-1.

[40] Merchant, A. and Alvarez, G. A., "Disk array models in Minerva," Tech. Rep. HPL-2001-118, Hewlett-Packard Laboratories, 2001.

[41] Ousterhout, J. K., Da Costa, H., Harrison, D., Kunze, J. A., Kupfer, M., and Thompson, J. G., "A trace-driven analysis of the UNIX 4.2 BSD file system," in *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pp. 15–24, ACM, December 1985.

[42] Poess, M. and Floyd, C., "New TPC benchmaks for decision support and web commerce," in *ACM SIGMOD Record*, vol. 29, pp. 64–71, December 2000.

[43] Raab, F., Kohler, W., and Shah, A., "Overview of the TPC benchmark C: The order-entry benchmark," tech. rep., Transaction Processing Performance Council, http://www.tpc.org/tpcc/detail.asp, December 1991.

[44] Ramakrishnan, K. K., Biswas, P., and Karedla, R., "Analysis of file I/O traces in commercial computing environments," *Performance Evaluation Review*, vol. 20, pp. 78–90, June 1992.

[45] Roselli, D., Lorch, J. R., and Anderson, T. E., "A comparison of file system workloads," in *Proceedings of the USENIX Annual Technical Conference*, 2000.

[46] Ruemmler, C. and Wilkes, J., "An introduction to disk drive modeling," *IEEE Computer*, vol. 27, pp. 17–29, March 1994.

[47] Sarvotham, S. and Keeton, K., "I/O workload characterization and synthesis using the multifractal wavelet model," Tech. Rep. HPL-SSP-2002-11, Hewlett-Packard Labs Storage Systems Department, 2002.

[48] Shriver, E., Merchant, A., and Wilkes, J., "An analytic behavior model for disk drives with readahead caches and request reordering," in *Proceedings of SIGMETRICS*, 1998.

[49] Sreenivasan, K. and Kleinman, A. J., "On the construction of a representative synthetic workload," *Communications of the ACM*, vol. 17, pp. 127–133, March 1974.

[50] Standard Performance Evaluation Corporation, "Current SPEC benchmarks."

[51] Thekkath, C. A., Wilkes, J., and Lazowska, E. D., "Techniques for file system simulation," *Software—Practice and Experience*, vol. 24, pp. 981–999, November 1994.

[52] Uysal, M., Merchant, A., and Alvarez, G., "Using MEMS-based storage in disk arrays," in *Conference on File and Storage Technology (FAST'03)*, pp. 89–102, USENIX, mar 2003.

[53] Varma, A. and Jacobson, Q., "Destage algorithms for disk arrays with non-volatile caches," *IEEE Transactions on Computers*, vol. 47, February 1998.

[54] Veitch, A. and Keeton, K., "The Rubicon workload characterization tool," Tech. Rep. HPL-SSP-2003-13, HP Labs, Storage Systems Department, Available from http://www.hpl.hp.com/SSP/papers/, March 2003.

[55] Wang, M., Ailamaki, A., and Faloutsos, C., "Capturing the spatio-temporal behavior of real traffic data," in *Performance 2002*, 2002.

[56] Wang, M., Madhyastha, T. M., Chan, N. H., Papadimitriou, S., and Faloutsos, C., "Data mining meets performance evaluation: Fast algorithms for modeling bursty traffic," in *Proceedings of the 16th International Conference on Data Engineering (ICDE02)*, 2002.

[57] Wilkes, J., "The Pantheon storage-system simulator," Tech. Rep. HPL–SSP–95–14, Storage Systems Program, Hewlett-Packard Laboratories, Palo Alto, CA, December 1995.

[58] Wilkes, J., "Traveling to Rome: QoS specifications for automated storage system management," in *Proceedings of the International Workshop on Quality of Service (IWQoS'2001)*, pp. 75–91, Springer-Verlag, June 2001.

[59] Wunderlich, R. E., Wenisch, T. F., Falsafi, B., and Hoe, J. C., "SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling," in *Proc. of the 30th annual International Symposium on Computer Architecture (ICSA03)*, pp. 84–97, May 2003.

[60] Zhang, J., Sivasubramaniam, A., Franke, H., Gautam, N., Zhang, Y., and Nagar, S., "Synthesizing representative I/O workloads for TPC-H," in *Proc. of the 10th International Symposium on High Performance Computer Architecture (HPCA10)*, Feb 2004.

VITA

Zachary Kurmas was born in Flint, Michigan, on 18 March, 1975. He received a B.A. in Mathematics and a B.S. in Computer Science from Grand Valley State University in 1997. That fall, he began graduate studies at Georgia Tech.

During the summer of 2000, he had the opportunity to work with the Storage System Department at HP Labs. Under the direction of Dr. Ralph Becker-Szendy and Dr. Kimberly Keeton, he began researching automatic methods of generating synthetic I/O workloads.

He continued his storage systems research at Georgia Tech; and in 2004, under the supervision of Dr. Kishore Ramachandran, he completed his Ph.D. dissertation entitled "Generating and Analyzing Synthetic Workloads using Iterative Distillation".