Thesis Proposal: Automatically Synthesizing Representative I/O Workloads

Zachary Kurmas

April 25, 2003

Abstract

I have developed a tool (the Distiller) that automatically searches for those workload characteristics (also called attribute-values) that two block-level workloads must share in order to have the same behavior on a given storage system. These characteristics can then serve as the basis for representative synthetic workloads with which storage systems can be evaluated. Furthermore, the ability to identify those workload characteristics that determine the behavior of a storage system will provide insight into how workloads and systems interact. This may help us develop better disk array models, hardware designs, firmware policies (e.g., cache-replacement, prefetching, and re-order algorithms), and disk array configuration/management software.

The immediate contribution of the Distiller is that it will make the generation of representative synthetic workloads feasible. Currently, most designers use traces of real-world workloads to evaluate storage systems. These traces are accurate samples, but are also large, inflexible, and difficult to obtain (often for non-technical reasons such as privacy). Synthetic workloads address these disadvantages; however, it is considered prohibitively difficult to generate a synthetic trace that is similar to the workload on which it is based. The difficulty of generating representative synthetic workloads is, in part, a result of our current inability to specify precisely which workload characteristics have largest effects on the behavior of the disk array. Although we know at a high level what properties (e.g., locality and burstiness) have a very significant effect on performance, in many cases, we don't know how to most effectively quantify them.

My tool addresses this difficulty in two ways: First, it automatically searches through a library of known attributes (i.e., specific methods of analyzing a workload trace) until it finds a set of attributes whose corresponding attribute-values specify a synthetic workload representative of the workload on which it is based (i.e., any two workloads with the same values for the chosen attributes have approximately the same behavior). Second, if no known combination of attributes specifies an acceptably accurate synthetic workload, the Distiller provides hints about how to develop a more suitable attributes.

In this document, I first discuss the need for such a tool. Then, I describe the algorithms and ideas on which the Distiller is based. Finally, I discuss my method of evaluating the correctness and contributions of the Distiller.

To demonstrate that the Distiller functions as described, I propose using it to find attributes for about twenty-five target workloads. Twenty of these target workloads will be artificial workloads produced using workload models based on attributes in the Distiller's library. By using attributes in the Distiller's library, I can ensure that the Distiller will be able to find a set of attributes that specifies a synthetic workload indistinguishable from the target workload. The remaining five target workloads will be traces of real

workloads including an Open Mail server, the TPC-C and TPC-H benchmarks, a file system, and a trace of the specWEB benchmark.[1] Because these workloads are not generated based on attributes in the Distiller's library, I cannot guarantee that the Distiller will be able to produce a highly representative synthetic workload. Instead, I use these workloads to demonstrate how the Distiller can assist the development of at least one new attribute and corresponding generation technique that allows us to more accurately synthesize the workload in question.

To demonstrate that the Distiller is useful in practice, I propose two sets of experiments: First, I propose demonstrating that the synthetic workloads specified by the Distiller lead to more accurate evaluations of hypothetical situations (e.g. "what happens if the arrival rate increases by 10%") than simply modifying a workload trace in an ad-hoc manner. Second, I propose demonstrating that the synthetic workloads specified by the Distiller can be effectively used to help direct certain disk array configuration decisions, such as whether to add more cache, or whether to change the amount of prefetching.

Finally, I will present any insights into the interaction between workloads and disk arrays. The Distiller makes it practical to find the most "important" attributes for a large number of workloads and disk arrays. By comparing the attributes chosen for different workloads and disk arrays, I hope to obtain insights that may be useful to other researchers, such as those who develop disk array models. For example, I hope to determine whether there exists a single set of attributes (and hence, a single workload model) that is suitable for all workloads on a given disk array configuration. In theory, such a set exists; however, it may be too complicated to be used in practice. I also will analyze how the accuracy of a synthetic workload is affected by the precision with which attribute-values are measured. This will help us learn about the trade-off between the accuracy and complexity of synthetic workloads.

1 Introduction and Motivation

I have developed a tool (the Distiller) that automatically searches for those workload characteristics (more formally called *attribute-values*) that two block-level workloads must share in order to have the same behavior on a given storage system. These characteristics can then serve as the basis for representative synthetic workloads with which storage systems can be evaluated. Furthermore, the ability to identify those workload characteristics that determine the behavior of a storage system will provide insight into how workloads and systems interact. This may help us develop better disk array models, hardware designs,firmware policies (e.g., cache-replacement, prefetching, and re-order algorithms), and disk array configuration and management software.

Section 1.1 discusses the trade-offs between synthetic workloads and traces of real workloads and explains why the ability to efficiently generate representative synthetic workloads represents a large improvement in our ability to easily evaluate storage systems. Section 1.2 explains why the generation of synthetic workloads is currently prohibitively difficult. Sections 1.3 and 1.4 list the contributions my dissertation will make to the field and how I will demonstrate each contribution. Finally, section 1.5 provides an organizational overview of this document.

1.1 Motivation

The behavior of most computer systems — especially large storage systems, such as disk arrays — is heavily dependent upon the choice of workload. A disk array will behave quite differently when reading data from randomly chosen blocks than when repeatedly writing the same block of data. As depicted in figure 1, the first workload (the top workload, in which the data is spread uniformly over the disk) will utilize most of the array's components and will tend to have long response times; whereas, the second workload (the bottom workload, which repeatedly reads a single sector) will exercise primarily the cache

[1] Assuming, of course, that I receive permission to use these traces.

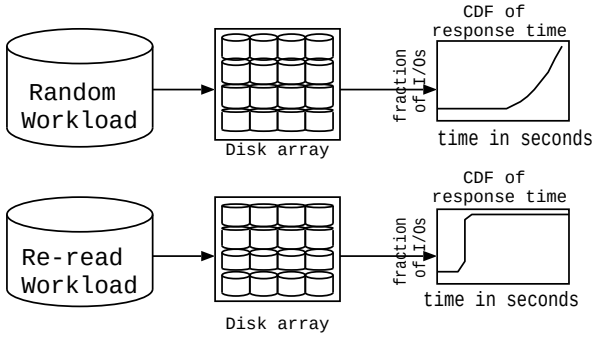

Figure 1: This figure is an example of how two different workloads have very different behavior. In this case, the behavior in question is the distribution of response times.

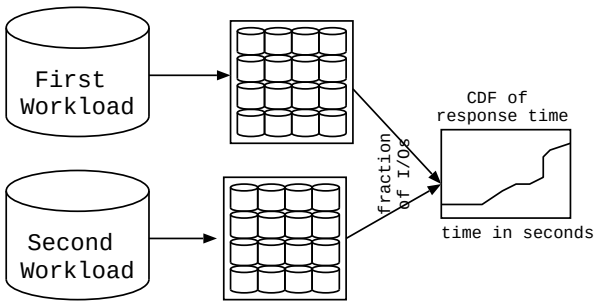

Figure 2: Two representative workloads will have the same behavior. In this case, the behavior in question is the distribution of response times.

and have very short response times. (In figure 1 the percentage of I/Os with long and short response times is shown using a cumulative distribution function, or *CDF*.)

Consequently, to accurately evaluate a system, we must use one or more workloads that will stress the system under test (SUT) in the same way the SUT will be stressed by the users. When evaluating a system, we want to use a workload that represents the expected production workload as accurately as possible. Ferrari calls the degree of this accuracy the *representativeness* of the test workload [15]. Henceforth, we will call the expected production workload we wish to model the *target* workload. In practice, an evaluation workload perfectly represents the target workload (with respect to some evaluation) if using the evaluation workload will cause the user reach the same conclusions she would have reached had she used the target workload.

Evaluation workloads are either *real* or *synthetic* [16]. A real workload is an actual production workload generated as a result of actual working conditions. Evaluators can obtain real workloads by performing the evaluation in the actual working environment; however, evaluators usually collect a trace of the real workload and replay it in an experimental, controlled environment. A synthetic workload is a workload specifically created to have a workload for study. Synthetic I/O workloads are frequently generated by choosing the values of I/O request parameters randomly from a set of distributions that are very carefully defined so as to maintain some given set of attribute-values. Example attributes for a block-level I/O workload include the mean request size, and the distribution of interarrival times. Example attributes for a file-level workload include the distribution of operations (e.g., open, close, read, write). The representativeness of a synthetic workload is determined by the set of attribute-values maintained. In other words, synthetic workloads tend to behave like those real workloads that have the same attribute-values.[2]

Workloads are most often studied by collecting and replaying a trace of the workload. Because they are exact copies of actual production workloads, traces of real workloads are more representative than synthetic workloads; however, as described in [16], the collection and use of real I/O workload traces possesses a number of limitations:

1. Traces can be quite large. 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 (10s of GB or more, depending on the length of the trace, the size of the traced

[2]We use the term *attribute* to name some type of workload measurement. Several example attributes are given in the paragraph above. The term *attribute-value* or *characteristic* refers to a specific measurement of a specific workload. For example "mean request size" is an attribute, whereas "mean request size of 8KB" is an attribute-value.

	Real Workload Trace	**Synthetic Workload**
Size	large	compact
Flexibility	none	attribute-values can be modified to represent hypothetical changes to workload.
Privacy	trace must be sanitized	compact representation already sanitized
Accuracy	perfect	approximate

Table 1: Comparison of workload traces and synthetic workloads.

system, the I/O activity during the traced period, and how much information is captured in the trace).

2. Traces are difficult to obtain, often for non-technical reasons. Any trace collection method has the potential to compromise privacy. In addition, the process of collecting traces can degrade performance. As a result, system administrators are reluctant to permit tracing on production systems.
3. Traces make it difficult to establish statistical confidence in results. Because each trace represents a single workload, it can be difficult to distinguish among real performance characteristics of the system under test, anomalous behavior of the trace, and anomalous behavior of the trace playback software.
4. Traces do not support studies of hypothetical workloads. A real trace collected yesterday cannot account for changes over time (e.g., an increase in the arrival rate as a business grows). This concern is especially relevant when using traces to evaluate a new system that may not be in production for months or years. Therefore, evaluators will occasionally modify a workload trace to evaluate the effects of various hypothetical changes over time. Similarly, evaluators may make small modifications to a real workload in order to test the sensitivity of the system under test. The modified workload trace, by definition, is a synthetic workload.

Synthetic I/O workloads do not suffer from these limitations.

1. It is possible to concisely describe most synthetic workloads using the attribute-values on which the workload is based.
2. The data collection methods can be much less intrusive. System administrators need only collect the attribute-values on which the synthetic workloads are based. More importantly, these attribute-values are almost always aggregate data. They rarely, if ever, contain any data that could compromise privacy.
3. It is possible to establish statistical confidence by randomly generating several synthetic workloads using the same characteristics, but different random seeds. Such comparisons may also help identify anomalies.
4. The attributes on which a synthetic workload is based can serve as "knobs" that evaluators can adjust in order to evaluate the effects of hypothetical situations. For example, if a synthetic workload is based in part on the arrival rate, an evaluator can adjust the arrival rate attribute-value to produce a faster or slower workload.

Table 1 summarizes the advantages and disadvantages of synthetic workloads.

1.2 Problem Statement

Synthetic workloads have one big limitation that currently outweighs their many advantages: In general, we don't know what attribute-values a synthetic workload must contain in order to be representative.

Workload	Attribute Set					
	Naïve	Best OpenMail	Best Cello	Best TPC-C	Best TPC-H	Best Oracle
Open Mail	.61 (.18)	.07 (.22)			.52 (.47)	
Cello	.54 (.33)	.18 (.33)	.41 (.5)		.34 (.67)	
TPC-C	.20 (.31)	.21 (.32		.09 (.36)		
TPC-H	.68 (.28)	.68 (.28)			.12 (.5)	
Oracle						

Table 2: Quality of best known generators. For each element, the first number represents the fraction error of the workload generator. The value in parenthesis gives the size of the synthetic workload's compact representation relative to the original trace. This table is currently incomplete because I have not yet experimented with all five workloads. This will be complete before the oral presentation of my thesis proposal.

This means that we do not know which attributes we should use to describe a given workload. It is difficult to generate an accurate synthetic workload when we don't even know how to precisely describe what we are generating.

This would be a trivial problem if we could simply have the synthetic workload maintain the value of every imaginable attribute. Unfortunately, there are many attributes for which it is hard to synthetically generate a workload with a specific attribute-value. (See for example, Conte and Hwu's "Interreference temporal density" and "Interreference spatial density" [10].) Furthermore, this problem is exacerbated by the fact that the workload generation techniques can interfere with each other. For example, we can easily generate a synthetic workload with a specific distribution of location values. Likewise, we can generate a synthetic workload with a specific distribution of jump distance. However, I know of no non-evolutionary algorithm that can produce a workload with both distributions.

As a result, scientists have developed a variety of workload generation techniques (sometimes called workload models) that reproduce a small number of attribute-values (many reproduce only one attribute-value). These generation techniques are developed by hand to reproduce what the developers believe are the most important attribute-values (i.e., those that have the largest effect on the behavior under study) as accurately as possible — often to the neglect of the other attributes. Because this effort is mostly trial-and-error, it is very impractical. In addition, it appears that the most important attributes for one workload may not be the most important attributes for another. For example, the behavior of one workload may be dominated by its bursty arrival pattern, whereas the behavior of a workload with a constant arrival rate may be dominated by its very high cache hit rate. This means that the effort developing a generator for one workload and storage system may not simplify the development of a generator for another workload or storage system. As demonstrated by table 2, for each generation technique tested, there exists some workload for which the resulting synthetic workload is not representative. In addition, there are some workloads for which no known generation techniques produce a representative synthetic workload.

As explained in section 1.3, the Distiller helps overcome the limitations by (1) automatically searching for those attributes whose corresponding attribute-values have the largest effect on the behavior under study, and (2) choosing compatible generation techniques (i.e. techniques that don't interfere with each other) that produce synthetic workloads with those attribute-values. This allows scientists to more effectively utilize attributes and corresponding generators developed by others. When a trace of the target workload isn't available, we can first identify the most important attributes for a set of workload traces, then describe the target workload using the corresponding attribute-values.

1.3 My Contributions

This dissertation addresses several of the problems encountered when attempting to generate a representative synthetic workload. The contributions of my proposed dissertation are as follows:

1. **The Distiller:** I have designed, implemented, and will evaluate a tool, the Distiller, that automatically searches for those workload characteristics (also called attribute-values) that two block-level workloads must share in order to have the same behavior on a given storage system. These characteristics can then serve as the basis for representative synthetic workloads. Thus, the Distiller will improve our ability to generate representative synthetic workloads.

   Given some target workload, the Distiller automatically searches through the set of known workload attributes until either

   (a) it identifies a set of attributes that specifies a synthetic workload representative of the target; or,

   (b) it determines that no combination of known attributes will specify a synthetic workload representative of the target.

   A set of attributes together with a trace of the target workload "specifies" a synthetic workload because it specifies the attribute-values that the synthetic workload must share with the target workload. The Distiller searches for a set of attributes for which any specified synthetic workload has similar behavior when replayed on a disk array. One example behavior of interest is response time. We will often consider two workloads representative if they have the same distribution of response time when replayed on a disk array. (See section section 2.4 for a discussion on various metrics for representativeness.)

   The Distiller automates the tedious trial-and-error aspect of developing a representative synthetic workload, thereby making it more practical. In addition the Distiller evaluates attributes in an intelligent order that avoids exhaustively searching all possible combinations. (The search space is currently small; however, we expect it to grow exponentially as new attributes and corresponding generation techniques are developed.) Furthermore, if no combination of known attributes produces a representative synthetic workload, the Distiller identifies the type of attribute-value that the synthetic workload does not share with the target. By partially describing a missing attribute, the Distiller helps direct the development of new attributes, when necessary and, therefore, helps reduce the tedium involved in searching for a useful workload generator. Section 3 discusses the design of the Distiller.

   I will evaluate the Distiller by showing that it can correctly find the desired attributes for at least twenty workloads on each of three disk arrays (the FC-60, FC-30, and an ordinary cluster of disks). Specifically, I show that each target workload and resulting synthetic workload have a similar distribution of response time. Section 4 discusses my evaluation of the Distiller.

2. **Useful synthetic workloads:** I see two immediate uses for the synthetic workloads specified by the Distiller. First, they can be used in place of workload traces to help make high-level configuration decisions. Second, they may be more useful than ad-hoc modifications to a workload trace for evaluating hypothetical situations. Section 6 discusses the potential uses of synthetic workloads.

   First, a system administrator must make many decisions when configuring and maintaining a storage system. For example, for an FC-60, she must decide how much cache to put in the disk array, how much data to prefetch, and what the high-water-mark of the write-back cache should be (i.e., when the disk array should begin committing data in the write cache to disk). Because making high-level changes to the disk array, such as adding cache, is very disruptive and potentially time consuming, administrators often simulate the performance of the disk array given the proposed changes. We expect that the synthetic workloads specified by the Distiller can be used in place of workload traces

as input to the simulation, and that doing so will lead the administrator to the same configuration decision.

Second, most hypothetical situations are currently evaluated using ad-hoc modifications to existing workload traces. For example, to estimate the effects of increasing the arrival rate by 20% one could take a workload trace and multiply the arrival rate by .8. The problem is that the resulting trace is only 80% as long as the original trace. Other ad-hoc modifications have similar issues. The use of synthetic workloads addresses this issue because the specification for the synthetic workload (i.e. the attribute-values) serve as obvious means by which a workload can be modified in a meaningful manner.

3. **Analysis of attributes chosen:** The Distiller's ability to easily find the "important" attributes for many different workloads and storage system configurations makes it possible to analyze the similarities and differences between many sets of attributes. This will provide some insight into the interactions between a workload and a storage systems. This insight may potentially help us develop better disk array models, hardware designs,firmware policies (e.g. cache-replacement, prefetching, and re-order algorithms), and disk array configuration and management software. Section 5 discusses different analyses that may interesting and useful.
4. **New attributes / workload generation techniques:** I plan to design, implement, and evaluate several new attributes and corresponding workload generation techniques that will allow me to generate representative synthetic workloads for target workloads that cannot currently be synthesized (because no existing generation technique produces a representative synthetic workload). Section 7 discusses several potential new generation techniques.

I believe that the techniques used in the Distiller generalize and can be applied to many more measures of representativeness than are addressed by this thesis. For example, representativeness could be defined in terms of power consumption or reliability. Also, I believe my techniques can applied to systems other than storage systems. For example, it may be possible to modify my techniques to study the behavior of programs on a new chip design.[3]

Finally, I emphasize that the focus of this thesis is finding and evaluating methods of automatically choosing important workload attributes from a set of known attributes. My focus is not on developing new attributes. To the best of my knowledge, I am the first to automatically search through a list of known techniques in order to develop an accurate synthetic workload generator. Although many people have developed attributes and workload generation techniques, and several people have attempted to generate complete synthetic workloads, I am not aware of anybody researching methods of automatically selecting attributes and/or generators from a library.

I intend to limit the scope of this thesis to the research that is necessary for the Distiller to produce representative synthetic workloads and to the analyses of the attributes chosen by the Distiller. I plan to leave the following for future work:

- Research into how best to use representative synthetic workloads.[4]
- Research into the optimality of any kind including: set of attribute chosen, representation of the attribute-values, and the Distiller's search algorithm. (For this thesis, I am building something that works. I'll worry about improving it after I've finished my Ph.D.)

The research most similar to mine is that of Ganger. In [16], he evaluated several different generation techniques to show that "commonly applied assumptions (e.g., uniform starting addresses, Poisson arrivals) are inappropriate ..." and to demonstrate that no simple technique produced a highly representative synthetic workload. Since then, there has been considerable research into synthesizing various

[3]For example, one might be able to write a version of the Distiller that automatically searches for those attributes that determine on which architecture (e.g., CISC,RISC, VLIW) a program will run most effectively.

[4]Well, it turns out I have to do a little of this to satisfy my committee.

aspects of a workload, especially the arrival and access patterns [17, 18, 19, 20, 33, 26, 32] (I discuss these in section 2.3); but, relatively little work has focused on on generating entire workloads.

1.4 Evaluation and Stopping Conditions

Here I briefly describe my method of evaluating each of the above contributions.

1. **The Distiller:** I will evaluate the Distiller by using it to produce representative synthetic workloads for about twenty targets on three disk arrays. These target workloads will be artificial workloads generated using techniques based on attributes in the Distiller's library. As a result, the Distiller will be able to specify a representative synthetic workload for each target. The difference between the behavior of the disk array when replaying the target and synthetic workloads should be almost as small (I propose a factor 2) as the difference between the behavior of the disk array when replaying synthetic workloads generated using the same generator but different random seeds. (This is what Ganger calls the *randomness error* [16].) This task will be complete when the Distiller can successfully find generators (and the corresponding attributes) for all twenty target workloads on each disk array.

   I will also evaluate the Distiller by using it to produce synthetic workloads for five real targets: a trace of the Open Mail e-mail server, a trace of a UNIX file system, and traces of the TPC-C and TPC-H benchmarks. Because the necessary attributes may or may not be in the Distiller's library, I cannot guarantee the Distiller will produce a representative synthetic workload. When it cannot, I will use the hint provided by the Distiller to help develop at least one new attribute and corresponding generation technique. This task will be complete when each workload is either successfully synthesized by the Distiller, or I have implemented at least one generator that improves the Distiller's ability to synthesize the target in question.

2. **Useful synthetic workloads:** I will evaluate the usefulness of the synthetic workloads specified by the Distiller in two ways. First, I will show that they have the same predictive value as a trace of the target workload. Second, I will evaluate whether synthetic workloads are better at answering hypothetical questions than modified workload traces.

   To demonstrate the predictive value of the synthetic workloads produced by the Distiller, I will show that the HP's Pantheon disk array simulator will lead a system administrator to the same configuration decision when run using a trace of the target workload or a synthetic workload specified by the Distiller.

   I will evaluate each of the real workloads used to evaluate the Distiller. In addition, I will consider three configuration decisions:

   (a) Cache size
   (b) Prefetching behavior
   (c) High-water mark of the write back cache

   (Note that I have just received the Pantheon disk simulator. I am making an educated guess as to its capabilities. Upon learning more about Pantheon, I may change the specific configuration decisions slightly.)

   To evaluate the usefulness of synthetic workloads in evaluating hypothetical situations, I will compare the representativeness of a synthetic workload whose arrival rate is doubled to the representativeness of a workload trace in which the interarrival times are multiplied by .5, but the trace is repeated (so as to produce a trace of equal length as the target workload).

3. **Analysis of attributes chosen:** My choice of which analyses to perform and present will depend on what interesting patterns and phenomena I notice while running Distiller. However, section 5 lists several analyses I expect to have interesting results.

	I/O Workload				Closely Related Values		
I/O Number	**Operation Type**	**Location**	**Request Size**	**Arrival Time**	**Interarrival Time**	**Jump Distance**	**Run Length**
1	Read	1024	8192	0	NA	NA	1
2	Read	9216	8192	.001	.001	0	2
3	Read	17408	1024	.003	.002	0	3
4	Write	33792	8192	.004	.001	15360	1
5	Write	18432	2048	.009	.005	-23552	1
6	Read	20480	4096	2.66	2.57	0	2
7	Write	19456	1024	2.69	.003	-5120	1
8	Write	51200	65536	7.87	5.18	32768	1

Table 3: Example workload and closely related values.

4. **New attributes / workload generation techniques:** I will evaluate the performance of each new generation technique on at least twenty workloads using the technique described in section 3.2.3.

1.5 Outline

Section 2 provides background. Specifically, it provides definitions, discusses how the design of a disk array affects performance, surveys the existing generation techniques, and explains my method of quantifying representativeness. Section 3 describes the ideas and algorithms on which the Distiller is based. Section 4 proposes an evaluation of the Distiller that will demonstrate that the Distiller produces correct results. Section 6 proposes an evaluation of the usefulness of the synthetic workloads generated by the Distiller. Section 5 discusses several potentially interesting analyses of the attributes chosen by the Distiller. Finally, section 7 discusses potential new workload generation techniques that I may implement and evaluate, if necessary.

2 Background and experimental framework

This section provides useful background information and provides the details of my experimental framework. Section 2.1 specifies the workload model I use and defines several terms that help describe workload characteristics and generation techniques. Section 2.2 discusses the design of the FC-60 disk array and how its components affect the response time of the array. Section 2.3 surveys the existing attributes and corresponding I/O workload generation techniques that are part of the Distiller's library. Finally, section 2.4 specifies my methods of quantifying the representativeness of a synthetic workload.

2.1 Workload

A workload for a disk array is a sequence of individual I/O requests. Each request has four parameters:

- **Location:** The *location* parameter identifies the location of the data in the disk array. An I/O's location includes both a device number (which identifies either a physical disk, or some logical partition of the disk array) and an address on that device. This pair can either be presented explicitly using two values, or implicitly with one value.
- **Request Size:** The *request size* is the number of bytes requested by an I/O request.
- **Arrival Time:** The time at which a request 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 arrival of the previous request. The choice of whether to present arrival time or interarrival time is a matter of convenience because each set of values can be calculated directly from the other (assuming an "open" model).

- **Operation Type:** A request's *operation type* is either "read" or "write".

Table 3 contains an example workload. In this example, the device number and sector number of a request's address are combined into a single value. Table 3 also presents the jump distance and run length for the sample workload. These terms, defined in section 2.1.2, are used to more clearly define several workload attributes and generation techniques.

2.1.1 Open vs. closed model

The description of a workload presented in section 2.1 is an *open* model. In an open model, the exact issue time of each request is specified. It is either relative to the beginning of the trace (arrival time), or to the issue time of the previous request (interarrival time).

Another common workload model is a *closed* model. In a closed model, the issue time of an I/O is specified relative to the completion time of the last synchronous I/O issued by the current thread. Thus, this model includes the CPU time between I/O requests issued by the same thread.

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

For this thesis, I will use only the open model. In order to increase the generality of my solution, I will, to the extend possible, limit the amount of information use the Distiller. By limiting my workload model to the four basic I/O parameters, the Distiller will be able to handle any workload trace, not just those that also include process IDs, thread IDs, and information about which I/Os are asynchronous.

2.1.2 Other terms

I define here several common terms are useful for describing a workload and its properties.

Run: 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. For example, I/Os 1 - 3 in table 3 form a run of length 3. I/Os 5 - 6 form a run of length 2.

Jump Distance: The *jump distance* is the distance in the location address space from the end of one I/O to the beginning of the next. For example, in table 3, the jump distance between I/Os 3 and 4 is $33792 - (17408 + 1024) = 15360$. The jump distance between I/Os 4 and 5 is $18432 - (33792 + 8192) = -23552$. Two successive I/Os in a run (e.g., I/Os 1 and 2) have a jump distance of 0.

Burstiness: The arrival pattern of a workload is considered to be *bursty* if there some are periods of time in which many I/O requests are made, and other periods of time in which very few, if any, requests are made.

Off time: An *off time* is a long period of time during which the workload makes no I/O requests. In order to measure off time, the user must choose a threshold of inactivity. My default threshold is two seconds. Thus, in my research, an off time is any period of at least two seconds in which there are no I/O requests. In table 3, there is an off time of 2.57 seconds between I/Os 5 and 6 and an off-time of 5.18s between I/Os 7 and 8.

On time: An *on time* is the period of time between two off times. In table 3, I/Os 1-5 and 6-7 form two on times of .009s and .003 respectively.

Footprint: A workload's *footprint* is the set of location values used at least once by a workload. The footprint of the workload in table 3 is

$$[1024, 18432), [19456, 24576), [33792, 41984), [51200, 116736)$$

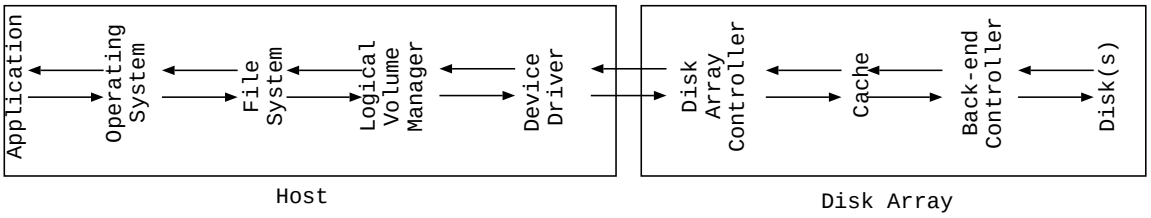

Figure 3: This figure shows the path of a typical I/O from the application through the disk array.

Notice that not every location accessed is the beginning location of an I/O. Location 2048 is accessed by the first I/O.

2.2 Disk arrays and performance

In order to better understand the difficulties of generating a representative synthetic workload and predicting performance, I discuss here the relationship among the disk array, a workload, and the performance of that workload on the disk array. Specifically, I will discuss the structure of the HP FC-60 disk array on which I have run most of my experiments. Most high-end disk arrays have a design similar to that of the FC-60. The FC-60 that I used contains sixty 36GB Seagate ST136403LC disks, spread evenly across six disk enclosures. It has two 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 512 MB of battery-backed cache (NVRAM) [5].

First, I will describe the path a typical read request from the application through the FC-60. (I will discuss write requests later.) This path is shown in figure 3. When an application makes an I/O request, it does so using a system call. The operating system handles that system call by handing it to the file system. If the request is not satisfied by the file system's cache, the file system hands the request down to the logical volume manager (LVM) which determines on which logical volume the requested data lies. The LVM then makes a request using the disk array's device driver. The device driver talks to the FC-60's disk controller over a fibre channel link. The disk controller first checks to see if the request can be serviced by the cache. If so, the requested data is sent back up the path to the application. Otherwise, the cache requests the data from the back-end controller. The back end controller determines exactly which disk or disks contain the data,[5] requests the data from the disks, assembles it if necessary, and returns the data to the cache. The requested data is then placed in the cache and send 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 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 particular disk that happens to be busy.

Most writes to an FC-60 have very low latency. The disk controllers have a 512MB write-back cache. This means that, the disk array almost always places the data of a write request into the cache and writes that data to disk at a later time.[6] This cache uses non-volatile RAM, which means that the data is safe, and the request is complete, as soon as the disk array's cache receives it. Requests for data stored in the cache have a very low latency because accesses to memory are at least two orders of magnitude faster than access to disks. Notice, however, that because the array must eventually copy the written data to disk (in other words, *de-stage* the data), the writes may contribute to the queuing delays of future

[5]If the disk array uses a RAID configuration, the data may be striped across several disks.

[6]The exception is when the write buffer is full. In this case, the write request is queued until is can be serviced.

requests.

If a request cannot be serviced by the cache, the disk controller must determine on which disk or disks the data is stored, and issue a request for that data. Identifying the physical location of data is not a trivial task. First, large disk arrays are often divided into several *logical units* (LUs). Each LU appears to the user as a separate storage device. On the FC-60, each physical disk is assigned to one logical unit.[7] In addition, each logical unit is organized according to some RAID configuration in order to provide the desired amount of redundancy and/or throughput [4, 5]. Organizing disks into a RAID either places copies of the data on several different disks (mirroring) or stripes the data across several disks.

The logical unit to which a set of data (e.g., a file system, or a database table) is assigned has a large effect on performance. For example, two database tables that are often accessed together should be assigned to different LUs, 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 an open problem currently under study by the Storage Systems Program at HP Labs [4, 5].

The disk controllers make several optimizations when servicing requests. First, they can carefully order requests to the disk drives so as to reduce bus contention, and minimize seek and queuing time. Also, the disk array can *prefetch* data. When the disk array prefetches data, it requests 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 also perform many of the same optimizations as the disk controllers. Modern disk drives have caches, reorder logic, and some can be configured to prefetch data. The interactions of a workload and a single disk have been extensively studied and are fairly well understood [25, 27]. Unfortunately, when many disks are placed together in a single device, the interactions become more complicated [24].

2.3 Summary of known techniques

This section discusses several of the attributes and corresponding workload generation techniques that researchers have developed in an attempt to to solve the workload generation problem. All of the techniques described here correspond to attributes that are (or will soon be) in the Distiller's library. As explained in section 3.1 a synthetic workload can be specified by the set of properties it shares with the workload on which it is based. A workload generator is a collection of generation techniques (i.e., algorithms) that each reproduces one or more high-level characteristics. In general, existing workload generation techniques do one of three things: Model the workload's source, reproduce a behavior-affecting pattern (attribute-value), or directly reproduce behavior. The next three subsections discuss these three approaches. Section 2.3.4 discusses techniques that combine two or more approaches

2.3.1 Model the source

Algorithms that model the source attempt to behave like the user and/or application that generated the target workload. For example, one could generate a sequence of http requests (either by modeling a user, or a web browser), issue those requests to a web server, and capture the resulting workload trace. Accurately describing user behavior (in this case, the sequence of http requests) is fundamentally the same problem as describing a workload (i.e., sequence of I/O requests) because the description of the user and/or application behavior implicitly contains a description of the workload characteristics to be reproduced; however, in some cases, the higher-level attribute-value may be easier to describe

[7] This is not necessarily true of all disk arrays. For example, the XP-512 allows multiple LUs to share a single disk.

and reproduce. For example, there have been several studies regarding the generation of http request patterns [11]. The challenge is that any given technique will work only for a very limited number of target workloads. (In this case a workload generated by a web server.) In some cases, it may be just as difficult to develop an accurate user- or application-level model as it is to generate the synthetic block-level I/O workload. In addition, the use of such a method requires detailed information about the source of the target workload and the data layout of the underlying file system. Such information is often not available. (None of the target workloads in this thesis contain source information; therefore, I cannot apply this method.)

SynRGen: SynRGen generates file system workloads by modeling the behavior of the user and application. SynRGen contains "micromodels" that simulate the I/O activity different applications [12]. For example, the micromodel of a compiler reads several files in their entirety (e.g., the .c and .h files), then creates and writes to another file (the .o file). SynRGen also contains models of different users. These models stochastically switch 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.

2.3.2 Reproduce the pattern

The second group of workload generation techniques explicitly attempt to reproduce some pattern. In other words, they measure some attributes of the target workload and generate a synthetic workload with the same attribute-values. These techniques are more general than modeling the source; however, they are only useful if the required set of attribute-values is reasonably simple. The attribute-values serve as a "compact representation" of the workload. It usually defeats the purpose of using a synthetic workload to have a set of attribute-values that is larger than the trace itself. (Exceptions include when when the synthetic workload is being modified to represent an expected future workload, and when the synthetic workload is being used for statistical confidence.)

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 inter-arrival times followed a Poisson distribution, location values followed a uniform distribution, and request sizes followed a normal distribution. They constructed an implicit distributions based on only a mean value and standard deviation, then drew values from this distribution regardless of its similarity to the 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 [16]. Using the distribution in the target workload produces a more accurate synthetic workload.

Unfortunately, distribution sampling rarely produces a representative synthetic workload because it does not measure and reproduce the correlations (relationships) that usually exist among request parameters [15]. 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. Note, however, that Bodnarchuk and Bunt successfully used this technique to generate a file system workload over NFS [6].

There are several ways to vary the accuracy of distribution sampling; however, increased accuracy comes at the cost of increasing the size of the synthetic workload's compact representation. A histogram is typically used to represent the target workload's distribution. The accuracy of the resulting synthetic workload depends upon the number of bins in the histogram. (A larger number of bins will produce a more representative workload.) Distribution sampling is most accurate when there is one bin for every possible parameter value. The trade-off for this increase in accuracy is an increase in the size of the synthetic workload's compact representation. The most accurate representation of interarrival time or location can be nearly as large as the workload itself.

Sampling from joint probability distributions can capture some correlations. For example, the corre-

	read	write
read	.75	.25
write	.80	.2

Table 4: Sample transition matrix.

lation between request size and interarrival time can be captured by drawing (request size, interarrival time) pairs from a two dimensional distribution. Similarly, the correlations between successive I/O request locations can be captured with a three dimensional joint probability distribution. (In other words, the values of three successive locations can be drawn at one time from a three dimensional distribution.)

The problem with joint probability distributions is that the number of necessary histogram bins increases exponentially with dimension. If there is a very strong correlation between two parameters, then there is a chance that a large percentage of the bins in the corresponding two dimensional histogram will be empty. In this case, the few full bins can be listed concisely.

State Transition Diagram: In many cases, the value of a request parameter depends heavily on the value of the previous request. For example, I find that, in practice, read requests tend to be followed by another read request; similarly, write requests tend to be followed by write requests. Thus, the probability that a particular request is a read is affected not only by the percentage of read requests in the workload, but also by the operation type of the previous I/O.

This property can be described using a state transition diagram. First, define two states: read and write. Next, calculate values for the matrix in table 4 as follows: The value in element (read, read) is the probability that, in the target workload, a read request is followed by a read request. The value in element (write, read) is the probability that a write request is followed by a read request. The values for the other elements are defined similarly. One can then generate a synthetic workload with this property by choosing a value for request $x + 1$ based on the operation type of request x and the probabilities given in the state transition diagram.

The value of this technique is limited when used alone. The size of the transition matrix grows quadratically with the number of states. Therefore, most request parameters will define a transition matrix that is much too large to be used practically. Instead, those who develop models often combine this technique with a conditional distribution. With a conditional distribution, the states do not correspond to individual parameter values but ranges of values.

Conditional Distribution: According to the principle of spatial locality, the current request's location is often near the previous request's location. This behavior can be reproduced in a synthetic workload by choosing values from different distributions depending on the location of the previous request. The distribution used when the previous location is 0 will have a high percentage of values around 0, whereas the distribution used when the previous location is 10GB will have a large percentage of values near 10GB. I call this technique a *conditional distribution* because the distribution used is conditional upon some value.

When using a conditional distribution, the distribution used is conditional upon some state. In the previous example, the states were "read" and "write"; however, the states can be defined in any arbitrary manner. In the case of request size, each multiple of 1KB can represent a separate state. Alternately, there can be three states: one for requests under 16KB, one for requests 16KB to 64KB, and one for requests above 64KB. Request sizes can even be assigned to states randomly. (However, I do not expect such a choice of states to be useful.)

After defining the states, one histogram is generated for each state. Each state's histogram presents the distribution some parameter over those requests corresponding to that state. In the previous example, one measured separate read/write distributions for the I/Os following reads and the I/Os following writes. In general any distribution can be measured: request size, interarrival time, or even some joint

distribution.

Finally, to generate the synthetic workload, choose some beginning state and draw a value from the corresponding distribution. Then calculate the new current state and draw a new value from the distribution corresponding to the new state. This iteration continues until all the desired I/Os have been generated. The new state can be chosen in two ways: It can be chosen directly based on previously chosen values, or a transition matrix can be used to determine the new state. The combination of a transition matrix and a conditional distribution is a *Markov model*.

As with the distribution sampling, a Markov model can be extended to generate several parameters at once by defining the states to be pairs of values. Similarly, it can be extend to generate correlations between I/O requests by defining the states to be sequences of successive request values. (Distribution sampling is simply a conditional distribution in which there is exactly one state.) However, like distribution sampling, increasing the number of states and dimensions increases the size of the compact representation exponentially. This growth is even more expensive than with the distribution sampling, because the addition of states adds entire histograms to the compact representation.

The low practical limit on the number of states poses the special challenge of how to best define the states. In the case of operation type there are only two possible states: read and write. However, when considering interarrival time, the choice of states is not nearly as obvious. If the user desires 10 states, one obvious scheme is to divide the range of interarrival times evenly into 10 segments (i.e., (0, 100ms), (100ms, 200ms), ..., (900ms, 1s)) Another is to divide the range according to the percentile (i.e., (0, .1), (.1, .2), ...(.9, 1.0)). These techniques also work for location and request size; however, because location values do not necessarily correspond to physical disk locations, the user may not want the states to be sets of contiguous location values. Furthermore, choosing sets of contiguous location values will not capture the behavior of different threads with interleaved requests. In section 7 I will examine more techniques for choosing states.

β-model: Wang et al. developed the β-model for generating arrival times [33]. This model is based on the "80/20" law for databases, which states that 80% of queries involve only 20% of the data. The β-model recursively allocates arrival times such that (when $\beta = 0.8$), during any given time interval, 80% of the arrivals happen in one half of the time interval and 20% happen in the other.

The parameter β is chosen based on the slope of the entropy plot.[8] Therefore, this technique is most suitable for workloads in which the entropy plot is linear. Hong and Madhyastha modified this technique slightly by using the multifractal spectrum to estimate β [20].

Multiplicative Wavelet Model: Sarvotham and Keeton implemented a different method of generating a bursty arrival pattern called the Multiplicative Wavelet Model (MWM) [26]. This model is based on the ideas of fractal Gaussian noise and wavelet transformations.

PQRS: Wang et al. have developed an attribute that "captures all the characteristics of real spatio-temporal traffic" [32]. In other words, this attribute-value not only captures the burstiness of the access pattern and the arrival pattern, but also captures 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 construction used by the β-model.)

Cluster-based trace synthesis: Hong, Madhyastha, and Zhang developed a generation technique that chooses several segments of a real trace to represent the entire trace. The algorithm then generates a complete synthetic trace from those segments [21]. Other scientists have used similar ideas to generate representative job mixes for use in evaluating entire computer systems.

[8] The entropy plot is a graph with entropy values on the y axis, and number of segments on the x axis. To calculate $f(x)$, divide the trace into x equal length segments, calculate the number of requests in each segment, then scale the values such that the sum is 1. The entropy value is then $-\sum_{i=1}^{i=x} p_i \log_2 p_i$ where p_i is the scaled value for segment i. The entropy value is minimized when all p_i are equal.

This 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 up to 10 seconds.) The algorithm then uses an agglomerate hierarchical clustering technique to groups the intervals into clusters. Next, one representative is chosen from each cluster. These representatives are the compact representation of the target workload. Finally, the algorithm produces a synthetic workload by concatenating copies of the cluster representatives.

This technique is very promising; however, its current clustering metric does not include spatial locality. Therefore, the number of workloads for which it will produce a representative workload is limited. However, as currently implemented, this clustering technique may be an excellent generator for parameters other than location.

2.3.3 Reproduce behavior

The third group of generation techniques do not attempt to reproduce any particular attribute-value. Instead, they attempt to generate each I/O such that is has a specific behavior (e.g., cache hit, or a specific response time.) I am unaware of any generation techniques that actually attempt to reproduce behavior; however, I provide here a hypothetical description of such a technique.

The input to this generator would be a distribution (or possibly a list) of response times. Then, for each I/O, the generator would consider the current state of the disk array, and choose the parameter values so as to produce an I/O with the desired response time. For example, if next I/O was to have a very short response time, the generator would choose an I/O that would be a cache hit. It would also adjust the interarrival time so as to avoid avoid queuing delays. Similarly, if a very long response time was desired, the generator would choose a cache miss and/or choose a very short interarrival time so as to cause a queuing delay.

Developing such a technique will be very difficult. Doing so requires a detailed model of the disk array that can very accurately predict the response time. There has been much progress in the development of disk array models, but not to the extent that the response time of an individual I/O can be accurately predicted [24, 29]. Second, it may not be possible to generate arbitrary combinations of response times. For example, extremely long response times may require a considerable queue length. 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.

Cache simulator: 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. This approach was used by Varma and Jacobson in a study of de-stage algorithms for disk arrays [30].

2.3.4 Combinations and compositions

The aforementioned categories of generation techniques are not mutually exclusive. Many generation techniques combine aspects of two or three categories. For example, the jump distance and run count techniques described below attempt to specifically generate low latency I/Os (reproduce behavior) by generating patterns of locality that are based on known application behavior (e.g., reading and writing data files from beginning to end). Likewise, Gomez's arrival time generator attempts to generate bursts of I/Os by modeling the ON/OFF behavior of many processes.

Jump Distance: Instead of choosing a location value explicitly, the jump distance method chooses the difference between the previous location and the current location. The principle of spatial locality tells us that the current request's location value is likely near that of the previous request. Using jump

distance, a generator can reproduce spatial locality without a separate state for each location value. This technique can be applied to any parameter; however, it is most useful for location.

Although computing a jump-distance attribute-value is straightforward, it is not obvious how to randomly choose a sequence of valid jump distances because the range of valid jump distances depends upon the current location value. For example, on a 9GB disk, if the current location is 0, only positive jump distances are valid; but, if the current location is 1GB, the range of valid jump distances is -1GB to 8GB. Obvious means of restricting the jump distance chosen affect the overall distribution of jump distances. One solution is to use the jump distance technique only for small jumps (e.g., less than 256MB). If the algorithm chooses a large, or invalid jump distance, then it chooses the location value is chosen from a distribution of valid locations. Such a solution must be implemented carefully in order to preserve the intended workload characteristics.

There are many algorithms for choosing a sequence of jump distance values. For example, we can use any of the distribution or Markov model techniques to measure and generate jump distances. Of course, the same concerns about number of bins and states apply.

Run Count: Another common technique is to generate "runs" of I/Os. (A "run" is defined in section 2.1.) Runs tend to have very low latency, especially on disk arrays that prefetch data.

Generators can easily produce such runs by choosing the run length and starting location, then generating the desired number of sequential I/Os. There are a number of methods for choosing the run length and starting location. The aforementioned distribution sampling and Markov model techniques can be adapted to measure and generate runs. Likewise, any technique that generates a location value can be modified to generate the starting location of a run.

It is interesting to note that the jump distance method also produces runs (provided the probability of a zero jump is not zero). However, the distribution of run counts generated in this manner will be Poisson. The run count method is more general, allowing for any distribution or pattern of run counts.

Gomez access pattern: Gomez and Santonja designed and implemented a location generation technique that combines elements of the distribution sampling, jump distance, and run count models. This model assumes that each I/O is associated with some process (i.e., that the workload trace also contains the ID of the process that generated the I/O), 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.
- spatially local (within 500 sectors) to the starting location for the current ON period.

The method used to choose a particular location is based upon 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 for 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 [18]. 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 specific distributions are defined based on corresponding mean values in the target workload (e.g., mean ON time, mean OFF time, etc.).

Read before write: In general, if a particular disk sector is both read from and written to by an application, it is read from before it is written to. This distinction is important on the FC-60 because of the write-back cache: If the disk array writes to the sector first, there is no compulsory cache miss because all writes are cache hits. In addition, subsequent accesses (both reads and writes) will be cache hits — provided that the cache does not evict the data. In contrast, if the disk reads from the sector first, the first read will be a compulsory cache miss. This difference in cache misses has a noticeable effect on performance for some workloads. To capture this behavior, I developed a generator that assures that both the target and synthetic workload have the same percentage of "read before write" requests.

Notice that this generator works as intended only if the cache never evicts the sector in question from the cache. I may need to extend this generator to account for evictions; however, at this time, I have not examined a workload for which such a generator is necessary.

2.4 Measuring representativeness

I use Ganger's method to determine the representativeness of a synthetic workload [16]. To determine if two workloads are representative (with respect to a given disk array and configuration), I replay a trace of each workload and compare the behavior of the disk array during each replay. This comparison method is shown in figure 5. In this figure the behavior quantified using the distribution of I/O request latency.

There are many ways to quantify the representativeness of a workload. By definition, two workloads are representative of each other when they stress the disk array in the same way. I could measure representativeness according to this strict definition by instrumenting the various components of the disk array and comparing the results. For example, given two workloads, I could (among many other things) compare their cache hit rates, amounts of data prefetched, length of queues in the disk controller, transfer times for each I/O, and response times for each I/O. This, however, would be very difficult, and of questionable practical value. Instead, I measure the aspect of disk array behavior that is directly observable to the user: performance. For the most part, the typical user cares about the behavior of the disk array only to the extent that it influences the responsiveness of applications.[9]

For this thesis, I will use two measures of representativeness:

1. distribution of response time, and
2. utilization.

2.4.1 Distribution of response time

Using the first definition, I will consider two I/O workloads to be representative with respect to some disk array if they have a similar distribution of response time. I could use a simpler metric such as a mean response time, or total execution time to measure performance; however, these metrics do not adequately reflect the number of requests with long response times. Most users would prefer a disk array that provides a consistent 2ms response time to an array that provides a 1ms average response time, but frequently "hangs" for several seconds.

The difference between two distributions of latency can be quantified in several ways. I expect that the Distiller will work with any reasonable metric; however, I identify here two metrics of particular interest: the root-mean-square, and the relative-root-mean-square.

- **RMS:** My primary metric is the root-mean-square (RMS) of the horizontal distance between the workloads' distributions of latency. RMS is similar to the average horizontal distance between the two distributions; however, the horizontal distances are squared to emphasize larger differences.

[9] Users also care about other measures of performance, such as power consumption and reliability; however, I leave this for future work.

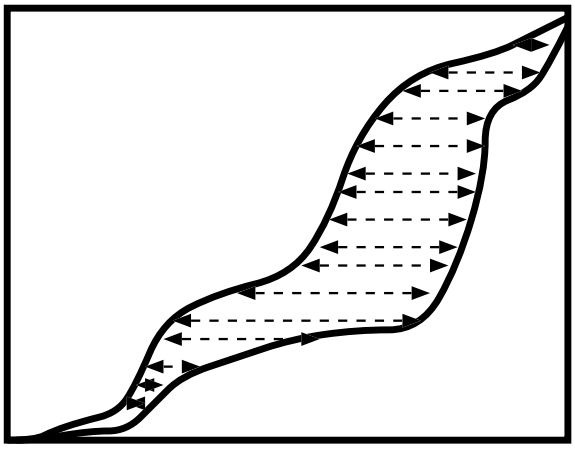

Figure 4: The arrows indicate the values that are squared and summed in the calculation of RMS.

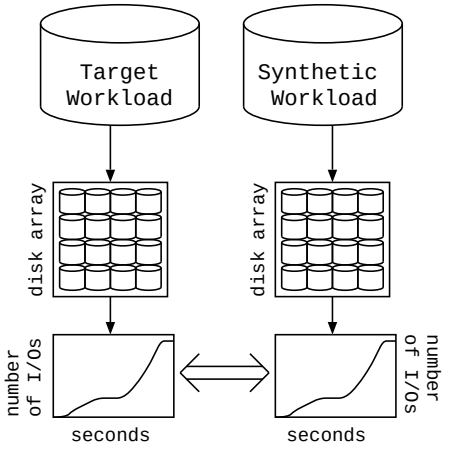

Figure 5: One method of evaluating representativeness is to replay two workloads on the same disk array and compare their distribution of latency.

Figure 4 illustrates the values that are squared and summed in the computation of RMS. Appendix A contains pseudo-code for an algorithm that computes the RMS of two distributions.

I chose this metric primarily because it is the metric used in the related work [25, 16, 17, 18, 26, 20]. In addition, use of the RMS metric is consistent with my desire to measure representativeness based on the behavior observed by the user. Because the top percentiles of a distribution function represent the number of high-latency I/Os, differences between the top percentiles of two distribution functions will make the largest contribution to the RMS value. These high-latency I/Os are precisely those I/Os that the user is most likely to notice.

Because the RMS metric emphasizes the differences of long-latency I/O requests, workloads with high mean response times will tend to have higher RMS differences than workloads with low mean response times. In other words, an RMS difference of .1 would represent a very close match between workloads with mean response times of 10ms, but would represent a very poor match when comparing two workloads with mean response times of 100μs. a In order to "normalize" the demerit figure across all workloads, we will often divide the RMS difference of two workloads by the target workload's mean response time. Dividing the RMS metric by the target workload's mean response time is done to make the results more understandable for humans; it will not have any significant effect on the operation of the Distiller.

- **rRMS:** I developed a similar metric called the relative-root-mean-square (rRMS). Instead of measuring the absolute horizontal distance, it measures the relative difference between distributions. The difference between 2ms and 4ms and the difference between 200 and 400 seconds affect the rRMS value equally.

  I designed this metric to account for the fact that the RMS metric emphasizes differences in the number of high-latency I/Os (e.g., request that require a seek) and de-emphasizes differences in low-latency I/Os (e.g., cache hits). This difference in emphasis is not an issue from the user's perspective; however, I occasionally desire a metric that better reflects the true behavior of the disk array.

2.4.2 Utilization

Using the second definition, I consider two workloads to be representative if they have the same utilization. I measure the utilization of a workload by comparing the number of I/Os completed during the trace period to the number of I/Os in a workload in which all interarrival times are zero, but the maximum number of outstanding requests is limited. In other words, I compare the target workload's I/O rate to the I/O of the workload generated by issuing requests fast as possible while maintaining reasonable limits on the number of outstanding I/O requests.

I choose this definition of utilization because it is the measure of utilization predicted by Eric Anderson's model [3]. By using his definition, I will be better able to evaluate the Distiller relative to existing technology.

2.4.3 Types of error

Ganger called the difference between the performance of two workloads on a given disk array the *total error* [16]. The total error quantifies the difference in behavior of a disk array when replaying two workloads (in my case, when replaying traces of a target workload and a synthetic workload). However, to evaluate the quality of the workload generator that produced the synthetic workload, one must consider two other types of error: the replay error and the randomness error.

Replay error: The *replay error* is the difference between two replays of the same workload. Disk arrays, operating systems, and networks are all very complicated and contain a small amount of non-deterministic behavior. Therefore, even two replays of the identical workloads will not have identical patterns of behavior. Specifically, a workload's distribution of response time will be slightly different each time it is replayed.

In addition to the non-deterministic aspects of the system, a disk array's cache replacement policy can contribute to a workload's replay error. The FC-60's cache replacement policy is very complicated and proprietary. The policy is designed to improve the hit rate over LRU, especially when the array detects a highly random, or highly sequential pattern. This means that I can neither completely flush the cache, nor put the cache in a known state before replaying a workload.

I use Ganger's method to quantify a workload's replay error [16]. First, I replay the workload several times. I then compute the "mean" of the synthetic traces by combining the response time distributions into a single distribution (i.e., I calculate the sum of the corresponding bins for each trace). The replay error is then the average RMS difference between each replayed workload and the "mean" workload.

Randomness error: The *randomness error* is the difference between two synthetic workloads generated by the same generator, but with different random seeds. As with the replay error, the randomness error is the average RMS difference between each differently seeded synthetic workload and the "mean" synthetic workload. Notice that this method of computing the randomness error (used by Ganger [16]) also includes replay error. To remove the replay error from this figure, one could replay each synthetic workload several times, create one "mean" workload for each random seed, then compute the randomness error of the "mean" workloads.

Synthesis error: Ganger defines the synthesis error as the difference between the total error and the randomness error. (Ganger doesn't differentiate between randomness and replay errors. Note that his method of calculating randomness error includes the replay error.) Specifically, he defines the synthesis error as max(0, total error - randomness error). If a synthetic workload's randomness error is larger than the total error, then the synthetic workload is statistically indistinguishable from the target workload.

For this thesis, we will use the synthesis error to quantify the quality of a synthetic workload. This metric is reasonable, however, only if the randomness error is sufficiently low. We have not yet determined a useful upper bound for randomness error. For all results in this thesis, unless stated otherwise, the sum of the randomness and replay errors will be at most twice the replay error, unless stated otherwise.

3 The Distiller

This section discusses the implementation of the Distiller. Section 3.1 carefully distinguishes between *attribute* and *attribute-value*, explains how attributes and attribute-values are used to specify a synthetic workload, and provides an overview of the Distiller's iterative loop. Section 3.2 introduces *attribute groups* and explains how the Distiller uses them. Section 3.3 presents the Distiller's complete algorithm.

3.1 Foundations

This section precisely defines *attribute* and *attribute-value*, explains how attributes and attribute-values are used to specify a synthetic workload, and provides an overview of the Distiller's iterative loop.

3.1.1 Attribute and attribute-value

An *attribute* is the name for some measurement of a workload. For example, "mean request size", "read percentage", and "distribution of location values" are all attributes. To be precise, an attribute is simply the name of a function whose domain is a workload trace. An *attribute-value* is an attribute paired with the specific measurement for a specific workload. For example, a mean request size of 8KB, or a read percentage of 68% are both attribute-values. If one views an attribute as the name of a function, f, then an attribute-value is the pair $(f, f(x))$ for some workload trace x.

Equating attributes with functions on workload traces places two important limitations on attributes: First, they must specify measurements of the workload trace only. The measurement specified by an attribute cannot depend on non-constant information not found in the workload trace. For example, "mean response time" is not a valid attribute because computing the mean response time requires knowledge of the specific disk array (and disk array configuration) on which the trace will be executed.[10] Second, an attribute must be fully defined. For example, "locality" or "burstiness" are not valid attributes because there are many different ways to quantify locality and burstiness. In contrast, the interreference temporal density [10] (see section 7.1.1) and the Hurst parameter of interarrival times are valid attributes.

3.1.2 How attributes specify synthetic workloads

In order for a synthetic workload to behave like the target workload on which it is based, it must share certain key properties with the target workload. For example, (given a typical disk array with a 256MB cache, and using response time as our measure of representativeness) a workload with a 64MB footprint will often have a much higher cache-hit rate and much lower mean response time than a workload with a footprint of several gigabytes. Thus, any algorithm that produces a representative synthetic workload (with respect to a typical disk array) must ensure that the footprint of the synthetic workload is similar to the footprint of the target workload on which it based. Almost any synthetic workload with a significantly different footprint size will have a different mean response time.

One method of specifying a representative synthetic workload is to give a complete list of these "important" properties. This list of properties is a set of attribute-values. An equivalent specification of a synthetic workload is a set of attributes and a target workload. From these, the corresponding set of attribute-values can easily be computed.

Not all attribute-values have the same effect on the resulting synthetic workloads. Consequently, we believe that it is not necessary to find every behavior-affecting attribute-value. Instead, we need only find those attribute-values with the largest effects on behavior. The next subsection provides an overview of how the Distiller finds these "most important" attribute-values.

[10] However, "mean response time on an FC-60 with 4 RAID-5 LUs,256MB of cache, and no prefetching" could be a legitimate (albeit very difficult to compute) attribute.

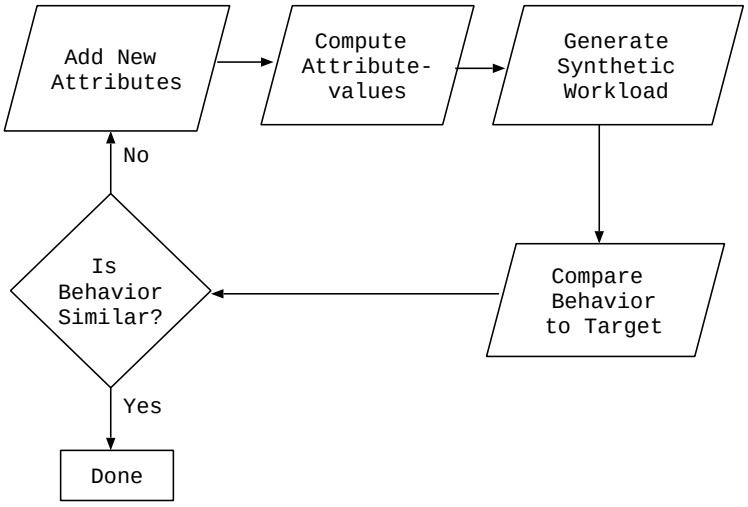

Figure 6: The Distiller iteratively adds attributes until it builds a set of attributes that specifies a representative synthetic workload.

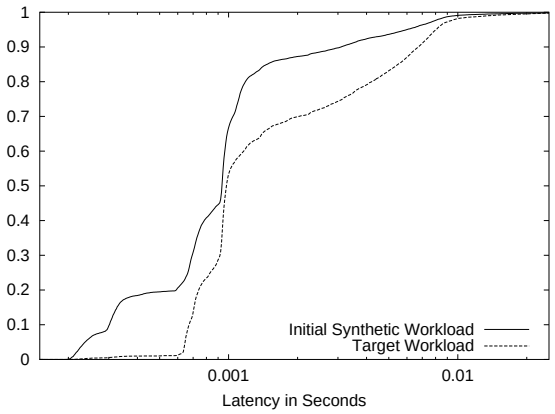

Figure 7: This figure shows that the OpenMail workload cannot be accurately synthesized using only the empirically observed distributions of values for the four I/O request parameters.

3.1.3 Basic idea of Distiller

The basic idea of the Distiller is to begin with a basic set of attributes — the set of distributions from which I/O request parameter values will be drawn (see section 3.3.1) — then iteratively add one attribute at a time until the synthetic workload specified by the set is sufficiently representative. Figure 6 shows this loop. With each new attribute, the Distiller first produces a synthetic workload based on the corresponding attribute-values. It then tests the representativeness of that synthetic workload (as explained in section 2.4 and shown in figure 5). This loop of adding attributes and evaluating the resulting synthetic workload continues until either

1. the difference between the behavior of the synthetic and target workloads is below some user-specified threshold, or
2. the Distiller determines that no set of attributes in the library will specify a representative synthetic workload.

As explained in section 1.2 , it is not feasible to simply include every conceivable attribute because the algorithms to generate a synthetic workload with a given attribute-value often interfere with each other. Hence, the complexity of generating a synthetic workload is (informally) related to the number of attribute-values maintained. Likewise, the size of a synthetic workload's "compact representation" is also related to the number of attribute-values maintained. Thus, for practical reasons, the Distiller searches for a small set of attribute that specifies a usefully accurate synthetic workload.

In order to have a reasonable running time, the Distiller must evaluate attributes[11] in an intelligent order. Our current evaluations of representativeness (see section 2.4) measure the behavior of a disk array while replaying a workload trace. The running time of each evaluation is on the order of tens of minutes. Thus, to be practical, the Distiller can only perform on the order of 1000 evaluations. Suppose the specification for a representative synthetic workload requires a set of six attributes. An

[11]The phrase "evaluate an attribute" really means "determine the extent to which adding an attribute to the current set of 'important' attributes increases the representativeness of the synthetic workload.

exhaustive search of possible attribute sets will become impractical when the library contains more than 15 attributes.

Furthermore, evaluating arbitrary sets of attributes is impractical because many pairs of attribute-values are highly redundant. For example, there are many different methods of quantifying burstiness and locality. In many cases, adding a second or third burstiness or locality attribute will do little to improve the representativeness of the specified workload. In section 3.3 we will see how the Distiller avoids evaluating redundant pairs of attribute-values by grouping attributes and evaluating entire groups with a single evaluation.

3.1.4 Target workload

We find that many of the concepts used by the Distiller make sense only in the context of a specific example. Therefore, as we explain the Distiller's algorithms, we will walk through an execution of the Distiller. This will provide a concrete example for each concept presented.

The target workload trace for this example is a 900 second trace of the workload created by the OpenMail e-mail application. For simplicity, we will be examining only one LU. The high-level characteristics of our baseline Open Mail workload are as follows: The workload contains highly randomized accesses using small I/O requests that are mostly (72%) writes. Over 90% of the requests have request sizes of 8 KB or less, with almost 50% of the requests being exactly 8 KB. The meta-data portion of the e-mail logical volume is frequently accessed, while the e-mail message (i.e., data) portion of the volume does not exhibit the same temporal locality. The complete workload is described in more detail in [22]. Our baseline trace contains 19,769 I/Os, with an average request rate of 22 requests per second, and an average throughput of 164 KB/s.

For this example, we will measure representativeness using Ganger's RMS method (see section 2.4). Specifically, the demerit figure for a synthetic workload will be quantified by dividing the RMS difference with the target workload by the mean response time. Our goal is to produce a synthetic workload with a demerit figure of at most .1.

3.2 Choose new attributes

During each iteration, the Distiller chooses an attribute to add to the set of "important" attributes on which it bases the synthetic workloads. Instead of evaluating attributes in an arbitrary order, we can estimate the maximum potential benefit of an entire group of attributes. This section describes how such groups are defined and how a new attribute is chosen.

3.2.1 Attribute Groups

We partition attributes into *attribute groups* according to the subset of I/O request parameters that must be considered when calculating the corresponding attribute-values. (Recall from section 2.1 that a workload is a sequence of I/O requests. Each I/O request contains four parameters: location, operation type, request size, and arrival (or interarrival) time.) When computing an attribute-value, it is often necessary to consider only a subset of the four I/O request parameters. For example, the mean request size of a workload can be calculated using only the request size of each I/O request. Likewise, most burstiness attribute-values can be calculated by considering only arrival times. Other attribute-values use the values of two or more different workload parameters. For example, the distribution of read request sizes must consider both the request size and operation type of each I/O.

Each attribute is a member of exactly one attribute group. The distribution of read request sizes is a member of the {request size, operation type} attribute group. It is not a member of the {request size} attribute group because it also considers operation type. Likewise, it is not a member of the {operation

{Location}	{Interarrival Time}
Distribution of Location	Distribution of Interarrival Time
Markov Model	Markov Model
Jump Distance Distribution	MWM
Jump Distance Markov Model	Wang's β-Model parameters
Run Count Distribution	Hurst parameter of arrival time
Run Count Markov Model	Coefficient of variance of arrival time
Inter-reference temporal distance	
Inter-reference spatial distance	
Parameters for Gomez arrival generator	

{Request Size}	{Operation Type}
Distribution of Request Size	Distribution of Operation Type
Markov Model	Markov Model

{Location, Request Size}	{Location, Operation Type}
Joint distribution of (Location, Request Size)	Join distribution of (Location, Operation Type)
Markov Model of (Location, Request Size) pairs	Markov Model of (Location, Operation Type) pairs
Conditional Distribution of Location based on Request size	Conditional distribution of Location based on Operation Type
Conditional Distribution of Request Size based on Location	Conditional distribution of Operation Type based on Location.

{Location, Interarrival Time}	{Location, Operation Type, Request Size}
Distribution of (Location, Interarrival Time)	Distribution of (Location, Operation Type, Request Size) triples
Markov Model of (Location, Interarrival Time) pairs	Markov Model of (Location, Operation Type, Request Size) triples
Conditional Distribution of Location based on Interarrival Time	Conditional distribution of Location based on (Request Size, Operation Type) pairs
Conditional Distribution of Interarrival Time based on Location	Conditional distribution of (Operation Type, Request Size) pairs based on Location.
Parameters to Wang's PQRS model	

Table 5: List of various partial generators.

Target I/O Workload				Initial Synthetic Workload				Additive Method Workload			
Op	**Location**	**Size**	**Arrival**	**Op**	**Location**	**Size**	**Arrival**	**Op**	**Location**	**Size**	**Arrival**
W	6805371	3072	0.050397	W	3339	16384	0.014152	W	3339	3072	0.014152
R	7075992	8192	0.762780	W	9076792	7168	0.336366	W	9076792	8192	0.336366
W	11463669	3072	0.789718	R	11657984	8192	0.428722	R	11657984	3072	0.428722
R	7051243	1024	0.792745	W	7194696	5120	0.429264	W	7194696	1024	0.429264
W	11460856	8192	0.793333	R	3485363	1024	0.429291	R	3485363	8192	0.429291
W	11463669	3072	0.808625	W	9007864	2048	0.439021	W	9007864	3072	0.439021
R	7049580	4096	0.808976	W	3082	8192	0.461442	W	3082	4096	0.461442
R	7050244	2048	0.809001	R	11665072	1024	0.773954	R	11665072	2048	0.773954

Table 6: Sample traces of target OpenMail workload, initial synthetic workload, and synthetic workload for the additive method.

type} attribute group because it also considers request size. Table 5 lists several attribute groups and example member attributes.

There are four workload parameters; therefore, there are 16 possible subsets of workload parameters. The empty set does not correspond to an attribute group. This leaves 15 possible attribute groups. The Distiller will mainly use those attribute groups corresponding to sets with one or two workload parameters (henceforth called *one-parameter*, or *two-parameter* attribute groups). Because attribute groups correspond to sets, order is not important. Thus {request size, location} and {location, request size} refer to the same attribute group.

3.2.2 Evaluating attribute groups

This section explains how the Distiller evaluates every attribute in an attribute group using a single evaluation. For sake of exposition, we will first consider specifically the {request size} attribute group.

Consider a synthetic workload in which the sequence of request sizes is identical to the sequence of request sizes in the target workload trace. The value for every {request size} attribute is identical to the target workload's corresponding attribute-values. Thus, this synthetic workload maintains every {request size} attribute-value. The Distiller uses this "perfect" {request size} attribute (i.e., the list of

Target I/O Workload				Subtractive Method Workload				Rotated Request Sizes			
Op	**Location**	**Size**	**Arrival**	**Op**	**Location**	**Size**	**Arrival**	**Op**	**Location**	**Size**	**Arrival**
W	6805371	3072	0.050397	W	6805371	16384	0.050397	W	6805371	8192	0.050397
R	7075992	8192	0.762780	R	7075992	7168	0.762780	R	7075992	1024	0.762780
W	11463669	3072	0.789718	W	11463669	8192	0.789718	W	11463669	8192	0.789718
R	7051243	1024	0.792745	R	7051243	5120	0.792745	R	7051243	3072	0.792745
W	11460856	8192	0.793333	W	11460856	1024	0.793333	W	11460856	8192	0.793333
W	11463669	3072	0.808625	W	11463669	2048	0.808625	W	11463669	3072	0.808625
R	7049580	4096	0.808976	R	7049580	8192	0.808976	R	7049580	1024	0.808976
R	7050244	2048	0.809001	R	7050244	1024	0.809001	R	7050244	8192	0.809001

Table 7: Sample traces of target OpenMail workload, and synthetic workloads used in the subtractive method.

request sizes in the target workload trace) as an upper-bound for the effect of every attribute in the {request size} attribute group.

There are several different methods by which the Distiller can use the "perfect" attribute to evaluate the {request size} (or any other) attribute group. The simplest method is the *additive* method. To use the additive method, the Distiller adds the perfect {request size} attribute to the current attribute list. In the OpenMail example, adding the perfect {request size} attribute replaces the randomly chosen request sizes with the sequence of request sizes in the target workload. Table 6 shows a sample workload trace for the target workload, the initial synthetic workload, and the synthetic workload generated using the additive method. After constructing the additive method workload shown in table 6, the Distiller compares the initial synthetic workload and the synthetic workload generated using the perfect {request size} attribute (shown in the center and right sections of table 6).

With our Open Mail example, adding the perfect {request size} attribute has almost no effect on the representativeness of the synthetic workload. This suggests that no {request size} attribute will have a large effect on the behavior of the disk array. The Distiller then excludes all {request size} attributes from further consideration.

In practice, the additive method does not always work well. Missing attribute-values can affect the behavior of synthetic workloads in unpredictable ways. The initial synthetic workload is based on only four attributes; therefore, it is likely missing many important attribute-values. Consequently, the behavior of this synthetic workload may be misleading.

To avoid some of the additive method's problems, the Distiller uses the *subtractive* method. The subtractive method estimates an attribute group's effects on behavior by removing all of the group's attributes from the target workload trace (as opposed to the additive method, which adds all of a group's attributes to an imperfect synthetic workload). In our OpenMail example, the Distiller applies the subtractive method to the {request size} group by generating a synthetic workload that is identical to the target, except that it contains no {request size} attributes. It does this, as it does for the initial synthetic workload, by generating request sizes independently at random from the empirical distribution. Values from the remaining request parameters are copied from the trace of the target workload. Such a workload is shown in the center of table 7. We will call this workload the "subtractive workload".

At this point, some people may be tempted to simply compare the behavior of the target workload to that of the subtractive workload. There is one problem: Not only does the subtractive workload in table 7 not maintain any {request size} attribute-values, it also does not maintain any {request size, location}, {request size, operation type}, or {request size, interarrival time} attribute-values. As a result, we cannot determine if any differences in behavior are a result of missing {request-size} attributes, or other attributes involving request size. To remove this ambiguity, we introduce the *rotated* workload.

In order to estimate the effects of removing only {request size} attributes, we generate a synthetic workload that is identical to the target workload trace, except the request sizes are "rotated". Such a workload is shown on the right of table 7. In this example, the request sizes are rotated forward by four I/Os. This workload maintains all attribute-values, except those in those multiple-parameter attributes groups involving request size.[12] Technically, the rotated workload does not maintain every {request size} attribute-value. Those attributes for which position is important will be affected. For example, rotating request sizes will slightly modify the Markov transition matrix of request sizes by producing a transition from 8192 to 3072 that is not in the original workload. In practice, we expect such differences to have a minor effect on the synthetic workload.

The rotated synthetic workload maintains the {request size} attribute-values as well as all attribute-values maintained by the subtractive workload. Therefore, the Distiller estimates the effects of all {request size} attributes by comparing the behavior of the subtractive workload and the rotate workload. We see in figure 10 that there is almost no difference in the response times of these two workloads. Thus, we conclude that a representative synthetic workload based on a trace of OpenMail need not maintain any {request size} attributes.

When we apply the subtractive method to the {location} attribute, we find that the difference between the two workloads is .16 (see figure 8); therefore, we conclude that there is some location attribute that will improve the current set of "important attributes". The next section discusses how the Distiller chooses a specific attribute from the {location} attribute group.

3.2.3 Choosing an attribute

Once the Distiller has chosen the attribute group from which the next attribute will come, it must choose a specific attribute. Currently, this is a matter of trial and error; however, there are several techniques that may possibly reduce the effort of the search.

Figure 8 indicates that some location attribute has a potentially significant effect on the resulting synthetic workload. At present, the Distiller simply iterates through the list of {location} attributes, until it finds a good candidate. We now explain how the distiller evaluates each candidate attribute.

The Distiller uses a variant of the subtractive method to evaluate each candidate attribute. In the case of the {location} attributes for our OpenMail example, the Distiller creates a workload that is identical to the target workload, except the location of each I/O request is chosen to maintain the {location} attribute under test as well as any {location} attributes previously added to the set of important attributes.. (In other words, we subtract all {location} attributes, except those under test, from the target workload.) This workload is similar to the subtractive workload for {location}, except the location values are generated in a way that maintains precisely the desired location attribute-values. This subtractive workload is then compared to the rotated location workload. The difference in behavior between the two workloads indicates the usefulness of the attribute-value. If the two workloads have very similar behavior (e.g., the demerit figure is within a given threshold), the Distiller adds the attribute to the set of "important" attributes. If the two workloads have very different behavior, the Distiller proceeds and evaluates other attributes. (Other details will be addressed in section 3.3.)

In the case of our OpenMail example, the Distiller first evaluates a Markov model of locations. This attribute divides the address space of the LU into 100 regions, and computes the transition matrix for the sequence of locations and the distribution of location values within each region. In figure 11 we see that a synthetic workload in which the location values maintain a 100 state Markov model behaves very much like a workload with the original, rotated sequence of location values. Therefore, the Distiller adds

[12]Specifically, the rotation workload does not maintain attribute-values in the following groups: {request size, location}, {request size, operation type}, {request size, interarrival time}, {request size, location, operation type}, {request size, location interarrival time}, {request size, interarrival time, and operation type}, and {request size, operation type, location, and interarrival time}.

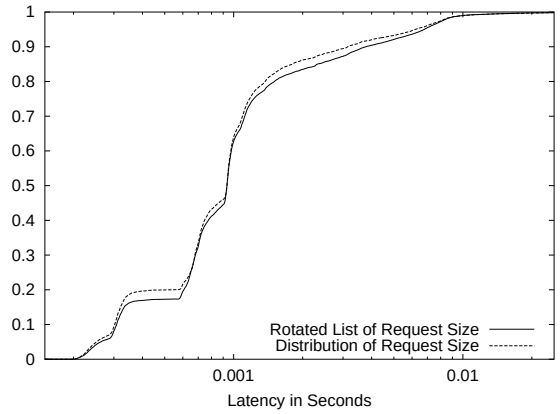

Figure 8: This figure shows that some {location} attribute is important.

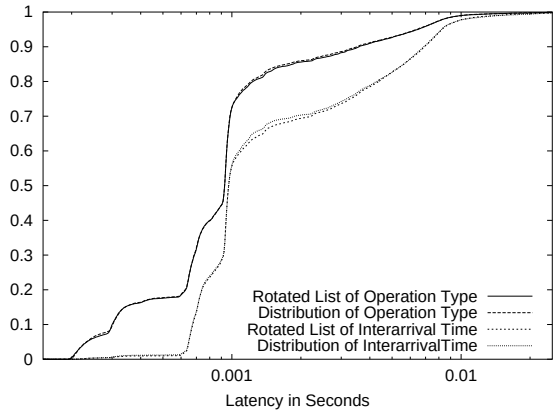

Figure 9: This figure shows that sequences of operation types and interarrival times drawn from an empirical distributions are representative of the corresponding sequences in the OpenMail workload.

this attribute to its list of important attributes.

3.2.4 Potential shortcuts

The Distiller currently iterates through the list of attributes in a pre-determined order. This works because, in practice, certain attributes tend to be used more frequently than others. By ordering the list of attributes according to frequency of use, the Distiller often finds the useful attribute quickly. Currently, we must set this order by hand. In the future, the Distiller may maintain a history of attribute use and order the list accordingly.

Also, for many attributes, we can identify conditions under which the attribute in question will or will not be effective. For example, consider a workload for which the Hurst parameter of arrival time is .5. Any set of numbers drawn independently at random from a distribution will have a Hurst parameter of .5. Therefore, a Hurst parameter attribute will only be useful when the target workload has a Hurst parameter significantly larger than .5. The Distiller could save time by not fully evaluating the benefit of adding a Hurst parameter attribute unless the corresponding attribute-value will be larger than .5. This calculation can be done in milliseconds, whereas a full evaluation requires replaying the trace requires minutes. I expect that the same principle can be used to avoid testing redundant attributes.

3.3 Complete Distiller Algorithm

In section 3.2, we explained the techniques used to choose an additional attribute for the set of "important" attributes. In this section, we present the remaining details.

3.3.1 Initial Conditions

The Distiller begins with an set of four attributes: one distribution for each I/O request parameter. The corresponding attribute-values are the empirically observed distributions of values for each parameter in the target workload trace. We choose empirical distributions as a starting point because every random number generator is based on some distribution. Beginning with an arbitrary or implicit distribution will potentially specify a synthetic workload based on incorrect, as opposed to missing, attribute-values.

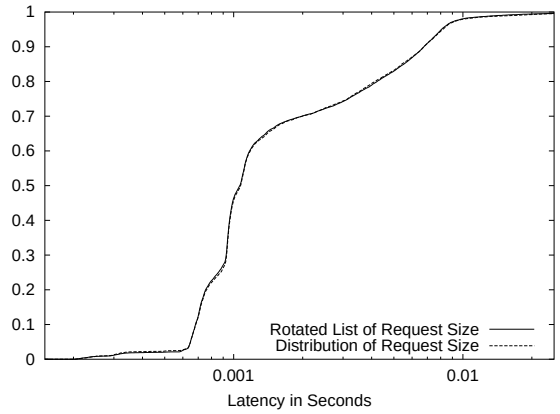

Figure 10: This figure shows a sequence of request sizes drawn from a distribution is representative of the sequence of request sizes in the OpenMail workload.

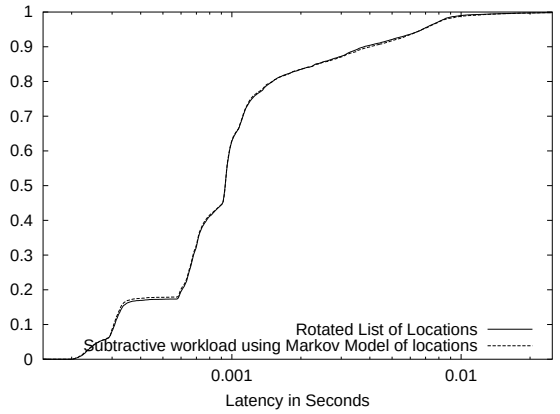

Figure 11: This figure shows that a sequence of location values generated by a Markov model is representative of the sequence of location values in the OpenMail workload.

In addition, Ganger demonstrated that the use of implicit (e.g., uniform or Poisson) distributions often led to unrepresentative synthetic workloads [16].

The Distiller's first steps are to generate a synthetic workload based on the initial set of attributes (shown in the center of table 6), then compute the demerit figure of that workload.[13] Upon doing this with the OpenMail workload, we find that the demerit figure is .56 (as seen in figure 7.) The demerit figure for the current set of attributes is larger than our threshold of .1; therefore, we must choose a new attribute to add to our set.

3.3.2 Phase 1

In phase 1, the Distiller searches for all of the important "single-parameter" attributes. These are the attributes in the {location}, {request size}, {operation type}, and {interarrival time} attribute groups.

First, the Distiller applies the subtractive method (as explained in section 3.2.2) to each single-parameter attribute group to determine whether any attributes from that group are needed. In the case of our OpenMail example, we see from figures 9, 10,and 8 that only the {location} attribute group contains important attributes.

For each chosen attribute group, the Distiller then uses the version of the subtractive method as described in section 3.2.3 to identify the important attributes in that group. When doing this, one of three things can happen:

1. *For some attribute, the difference between the subtractive and rotate synthetic workloads is very small.* In this case the Distiller adds the attribute in question to the set of important attributes and proceeds to the next attribute group.
2. *No attribute results in a small difference between the subtractive and rotate workloads; however, several attributes make a significant improvement over the current set of attributes.* In this case, the Distiller adds the attribute that results in the most improvement to the set of important attributes, then repeats the process on the same attribute group.

[13]Note: The example trace in table 6 was taken from a different OpenMail LU. This does not affect the principles illustrated by the table.

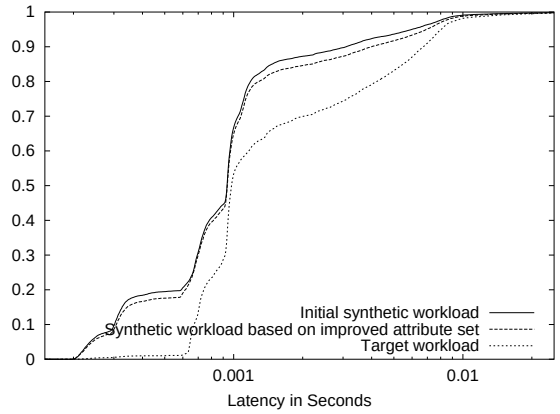

Figure 12: This figure shows that using a Markov transition matrix to choose location values improves the representativeness of the resulting synthetic workload; however, the improved workload is still not sufficiently representative.

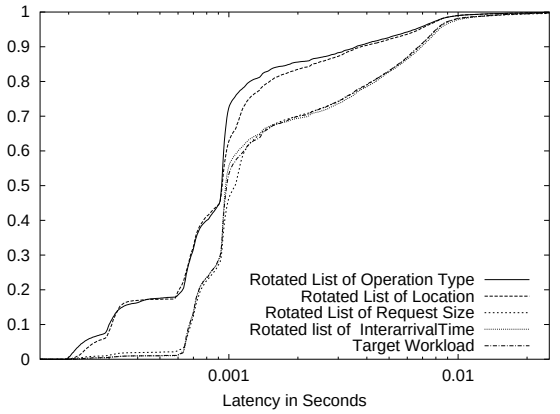

Figure 13: This figure shows that some {operation type, x} attribute and some {location, x} attribute have a significant effect performance; however, no {request size, x} and no {interarrival time, x} attributes have a significant effect on performance.

3. *No attribute results in a significant improvement.* In this case, the Distiller's library of attributes is insufficient. At this point the user has two choices: (1) Add more attributes to the library and re-start the Distiller; or (2) continue and find the best answer possible given the library's limitations.

We saw in section 3.2.3 that for our example OpenMail workload, the Distiller evaluated a Markov model of location values, and found that it resulted in situation 1: a synthetic workload with similar behavior to the rotated synthetic workload. Therefore, the Distiller added the Markov model to the set of important attributes.

After the Distiller finishes addressing each single parameter attribute, it evaluates the synthetic workload specified by the improved set of important attributes. Figure 12 shows the results for our OpenMail example. Because the demerit figure (.53) is not below the desired threshold, the Distiller proceeds to phase two.

3.4 Phase 2

In phase two, the Distiller addresses the two-parameter attribute groups. The Distiller begins phase two by comparing the rotation synthetic workload for each I/O parameter to the original workload trace. Recall from section 3.2.2 that the rotation workload for {request size} maintains all attribute-values, except those multiple-parameter attributes involving request size. We can see from figure 13 that there is little difference between the behavior of this workload and the target workload. Therefore, we conclude that no attributes in any {request size, x} attribute group has a large effect on behavior. In contrast, figure 13 shows that the behavior of the rotation workloads for operation type and location differ significantly from that of the target workload. Therefore, we conclude that some {operation type, x} attribute and some {location, x} attribute have a significant effect on behavior.

For each parameter selected above (in this case, operation type and location), the Distiller now identifies appropriate values for x. In the case of operation type, the Distiller determines which of the {operation type, location}, {operation type, request size}, and {operation type, interarrival time} attribute groups contain an attribute that should be added to the list of important attributes.

To evaluate the need for an {operation type, request size} attribute, the Distiller compares two workloads: A workload in which the operation type and request size values from the target workload

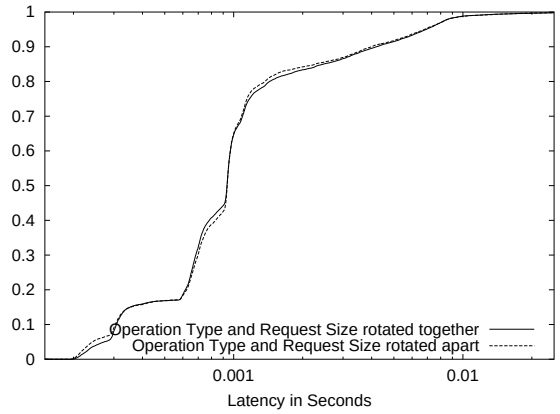

Figure 14: This figure shows that no {operation type, request size} attributes have a large effect on performance.

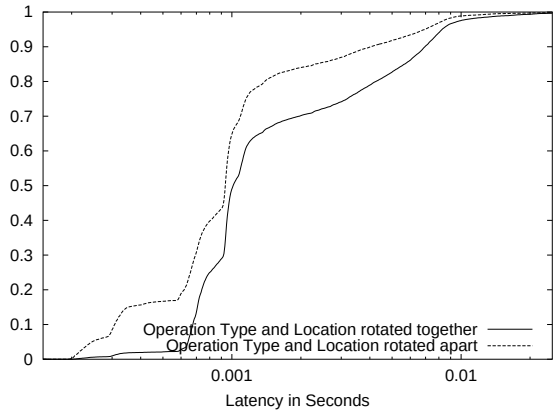

Figure 15: This figure shows that some {operation type, location} attribute has a significant effect on performance.

trace are rotated by the same amount, and a workload in which the values are rotated different amounts. We call these the "rotated together" and "rotated apart" workloads. These two workloads maintain the same set of attribute-values, except the "rotate together" workload maintains all {operation type, request size} attribute-values whereas the "rotate apart" workload does not. Therefore, differences in the behavior of these two workloads is an estimate of the importance of the {operation type, request size} attributes. Figure 14 shows that there are probably not any important {operation type, request size} attributes; however, figure 15 indicates that that there is an important {operation type, location} attribute.

Both the "rotate together" and "rotate apart" workloads are missing a number of attribute-values. Specifically, they are missing all {operation type, x} and {location, x} attribute values (with the exception that the "rotate together" workload maintains all {operation type, location} attribute-values). These missing attribute-values can introduce error and mask the true effects of {operation type, location} attributes. This is, in part, why the Distiller individually tests each {operation type,x} attribute group instead of simply assuming from figure 13 that the only important two-parameter attribute values are in the {operation type, location} attribute group.

When the Distiller has identified a performance-affecting two-parameter attribute group, it evaluates the candidate attributes as described in section 3.2.3. However, in this case, it uses information from phase 1 to narrow the set of attributes evaluated. Specifically, it will only evaluate attributes that do not affect the attribute-values for the corresponding single-parameter attribute groups. For example, suppose the Distiller chose a Hurst parameter for arrival time in phase 1. Furthermore, assume the Hurst parameter of arrival time for the target workload is .68. The Distiller will not consider in phase 2 any {arrival time, x} attribute for which workloads with the corresponding attribute-value have a Hurst parameter of .5.

In theory, such incompatibilities should be rare. However, most current workload generation techniques have "side effects". For example, the common method of generating an arrival pattern with the desired distribution is to simply draw values independently at random from the specified distribution. This algorithm does not preserve any "burstiness". Conversely, common methods of generating bursty arrival patterns do not preserve specific distributions of interarrival time.

For our example, the Distiller first evaluates an attribute that specifies separate Markov models of location values for read requests and write requests. Figure 16 shows that a synthetic workload that maintains the corresponding attribute-value has similar behavior to the synthetic workload in which

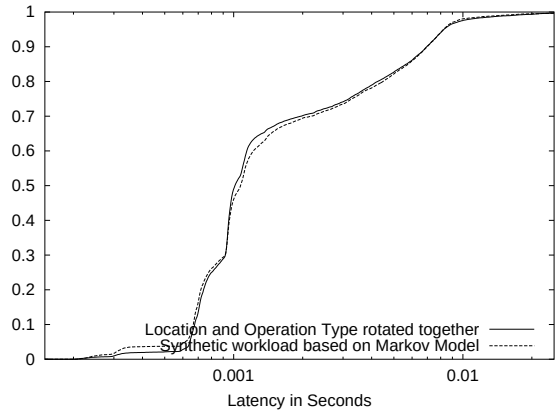

Figure 16: This figure shows that separate Markov transition matrices of location for read values and write values produces a representative pattern of operation type and location.

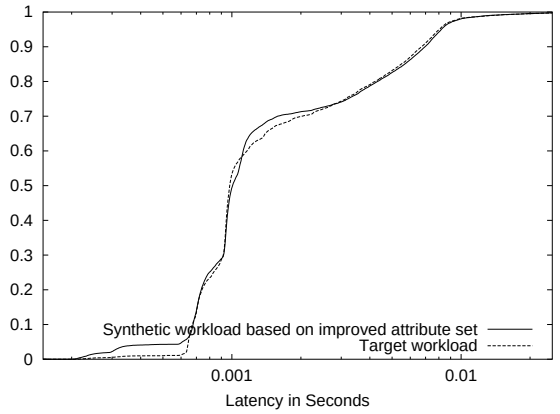

Figure 17: This figure shows that the improved synthetic workload (the one based on separate Markov transition matrices of locations for read requests and write requests) is representative of the target OpenMail workload.

operation type and location were rotated together. Therefore, the distiller adds this attribute to its list of important attributes.

After the Distiller has addressed each multiple-parameter attribute, it evaluates the synthetic workload specified by the improved set of important attributes. Figure 17 shows the results for our OpenMail example. Because the demerit figure(.08) is below the desired threshold, the Distiller terminates.

3.4.1 Additional phases

The next phases of the Distiller (if necessary) will search for important three-parameter and four-parameter attributes. The algorithms for these phases will be similar to those of phase 2. However, we have yet to encounter any workloads for which it is necessary to proceed beyond phase 2.

4 Evaluation of the Distiller

This section describes my method of evaluating the Distiller's ability to automatically find those attributes that specify a representative synthetic workload. As promised by contribution 1, I will evaluate the Distiller by running it using a series of at least twenty target workloads for which I know suitable attributes exist. These target workloads will be artificial workloads generated using techniques corresponding to attributes in the Distiller's library. (I will use artificial target workloads so that I am are assured that the Distiller will be able to find attributes that lead to a representative synthetic workload.) These artificial target workloads will be designed to represent the variety of workloads studied by the SSP and to exercise as many of the Distiller's features as possible.

I plan to run each of the experiments on three disk arrays: an FC-60, and FC-30, and a group of disks, affectionately known as the JBOD ("Just a Bunch Of Disks"). The FC-60 was described in section 2.2. The FC-30 is an older version of the FC-60. Its design is similar to that of the FC-60; however, the FC-30 holds only 30 disks, and its cache is only 60MB [2]. The JBOD is a collection of disks attached to a single SCSI bus. In addition, I may repeat each experiment several times using different random seeds in order to show that the Distiller is stable (in other words, that the quality of its answers are not heavily dependent upon the initial random seed).

For each disk array combination, I plan to present a table with the following information for each target workload:

1. A general description of the workload. (That is, a description of the workload's high-level characteristics that can be easily understood by a human.)
2. The representativeness (measured using RMS) of a workload produced using a naïve generator.
3. The representativeness (i.e., total error) of a synthetic workload produced using the generator chosen by the Distiller (i.e. specified by the attributes chosen by the Distiller).
4. The randomness and replay errors of this synthetic workload. (Either the randomness or replay error should be greater than the total error.)
5. The list of partial generators selected by the Distiller
6. The running time of the Distiller on this workload.
7. The size of the synthetic workload's compact representation compared to the size of the target workload.

It is not practical to quantitatively evaluate the quality of the Distiller's hints. Instead I will simply demonstrate how the Distiller processes several production workloads for which no known attributes specify a representative synthetic workload. I will show what hints the Distiller gives the user and demonstrate how those hints can be used to generate a new attribute.[14] For example, when given a trace of an Open Mail e-mail server, the Distiller identifies that no existing {location} attribute produces a representative disk access pattern. Furthermore, the distributions of latency for the synthetic workloads produced by existing generation techniques suggest that, although the synthetic workloads have a similar cache hit ratio, the distributions of seek times are quite different. As a result, I developed the Proximity attribute and generator. (Section 7 contains a full description Proximity.)

At this point, I have only just begun to search for an attribute suitable for the Open Mail workload. In addition to Open Mail, I also plan to synthesize the TPC-C and TPC-H benchmarks, a trace of a file systems at HP labs (cello), and a trace of the specWEB benchmark. Use of these traces is contingent upon receiving permission from HP labs.

5 Analysis of attributes

The Distiller's ability to easily find the "important" attributes for many different workloads and storage system configurations makes it possible to analyze the similarities and differences among many sets of attributes. This was not previously possible because there was no practical means of collecting a large set of attributes to study. Studying the attributes chosen for many different workloads and storage system configurations will provide some insight into the interactions between a workload and a storage systems. This insight may potentially help us develop better disk array models, hardware designs, firmware policies (e.g. cache-replacement, prefetching, and re-order algorithms), and disk array configuration and management software. This section discusses different analyses that may interesting and useful.

Do the attributes depend on the storage system and its configuration? The answer to this question is almost certainly "yes." Storage systems include many components designed specifically to address certain expected patterns in the workload. For example, storage systems have cache to most efficiently deal with the expected temporal locality. Some storage systems prefetch data to efficiently handle the expected spatial locality. Thus, the attributes chosen by the Distiller should reflect those workload patterns that the storage system is specifically configured to handle most efficiently.

A more interesting and potentially useful question is "**how different are the sets of attributes chosen for similar storage systems and configurations?**" For example, will the set of attributes

[14] Note: The Distiller cannot automatically generate new attributes. This task must still be done by hand.

chosen by the Distiller for two similar disk arrays (e.g., the FC-30 and the FC-60) differ by only one or two attributes, or will the two sets be nearly disjoint? Likewise, will small changes in the configuration of a disk array (e.g., different amounts of cache, or a different prefetching length) result in small changes to the attributes chosen by the Distiller? This information is important, because if similar configurations use similar attributes, then a single set of synthetic workloads can be used to evaluate several potential storage system configurations. If different configurations lead the Distiller to vastly different sets of attributes, then it may be much more difficult to use synthetic workloads to evaluate potential storage system configurations because each potential configuration will require a synthetic workload based on different attributes.

I plan to evaluate the distiller using at least three different storage systems. I will analyze the differences in attributes chosen for each storage system and attempt to relate any differences to the designs and configurations of each system. I will also determine which sets of storage systems and storage system configurations can be accurately evaluated using the same synthetic workloads.

Do the attributes depend on the workload? We expect that, given the same storage system configuration the Distiller will choose different attributes for different workloads. This will happen if the workloads utilize different aspects of the storage system's components. For example, we expect the Distiller will choose different attributes for a random workload whose footprint fits in the disk array's cache than for a highly sequential workload with a very large footprint. Likewise, the Distiller is unlikely to choose any type of "burstiness" attribute for a workload with a constant arrival rate.

We wonder whether the differences between workloads extend beyond those that are related to the set of disk array features exercised by each workload. Therefore, we ask whether **for each storage system, there is a single set of attributes that will specify a synthetic workload given any target workload?** I believe there is such a set for each storage system configuration. The more interesting question is whether that set of attributes is useful. We can naïvely obtain such a set of attributes by using the Distiller to find attributes for many workloads and taking the union of those attributes; however, a set of attributes obtained in this manner may be too large and complicated to support practically the generation of synthetic workloads.

I plan to take the attributes chosen for a number of different workloads and determine the following:

1. Which set of attributes is best overall. For each workload used as input to the Distiller, I will take the set of attributes, generate a synthetic workload for each other workload, and measure the representativeness of that workload. I will then order the different sets of attributes according to the mean error of the synthetic workloads.
2. Which set of attributes has the best worst-case. I will create synthetic workloads as above, but this time, I will order them according to the error of the synthetic workload that was least representative.
3. How big is the union of all attributes?
4. Can that union be used in practice to specify a synthetic workload? If not, can we easily find a smaller set that works well?

What is the trade-off between the precision (detail) of the attributes and the representativeness of the resulting synthetic workloads? Many attribute can be set to measure at various levels of precision. For example, the precision of a distribution attribute is determined by the number of buckets in the histogram used to represent the distribution. Likewise, the precision of a Markov model depends on the number of states. As the precision of an attribute decreases, so does the size of the representation of the corresponding attribute-value.

I will measure the trade-off between the precision of the attributes and the accuracy of the resulting synthetic workloads. I plan to measure this trade-off for only four or five workloads. This is not the focus of my thesis. My goal here is only to highlight the existence of the trade-off and provide preliminary results to see if there is any interesting future work in this area.

Can we assign a "percent contribution" to each attribute or attribute group? Given any set of attributes, the user has the option of removing an attribute, thereby trading the size of the attribute-values' representation for the accuracy of the resulting synthetic workload. We would like to investigate methods of ranking attributes by "importance". One measure of an attribute's importance is the the effect of its on the accuracy on the resulting synthetic workload. Understating general trends of importance among attributes will help us design better algorithms for the Distiller and make better space/accuracy trade-offs for synthetic workloads.

How do attributes and/or attribute-values change over time? We wonder if attribute-values change predictably over times. For example, given traces of a file system captured in 1992 and 1996, could we generate a trace representative of the same file system in 2002? We will closely examine the set of attributes chosen by the Distiller for the 1992, 1996, 1999, and 2002 cello file system traces. If the set of attributes remains constant over time, we will then attempt to predict how the attribute-values change.

6 Useful synthetic workloads

In this section, I discuss two evaluations of manageable scope that demonstrate the potential usefulness of the synthetic workloads specified by the Distiller. First, I determine the usefulness of these synthetic workloads in evaluating potential disk array configurations. Next, I determine whether synthetic workloads can more accurately evaluate hypothetical situations than ad-hoc modifications to workload traces.

Ganger [16] and others have suggested many potential uses for synthetic workloads and/or the attributes that specify them (see also [3]). However, because it is not currently feasible to generate representative synthetic workloads, few, if any, of these ideas have been implemented. Furthermore, the implementation of almost all of these ideas is outside the scope of this thesis. For example, I do not have the knowledge and experience to build a disk array model based on the attributes that specify a synthetic workload. The evaluations discussed below are are two I believe are of reasonable scope.

6.1 Evaluation of potential disk array configuration

A system administrator must make many decisions when configuring and maintaining a storage system. For example, for an FC-60, she must decide how much cache to put in the disk array, how much data to prefetch, and what the high-water-mark of the write-back cache should be (i.e., at what point the disk array should begin committing data in the write cache to disk). Because making high-level changes to the disk array, such as adding cache, is very disruptive and time consuming, administrators often simulate the performance of the disk array given the proposed changes. We expect that the synthetic workloads specified by the Distiller can be used in place of workload traces as input to the simulation, and that doing so will lead the system administrator to the same configuration decision.

We will consider three design decisions: the amount of cache in the disk array, the length of prefetching (if any), and the high-water mark of the write back cache.

- **Cache:** Data that is in the cache can be returned to the application very quickly because the disk array need not wait for any physical operations to complete (e.g., seeks and data transfer to or from the magnetic media). Up to the point that the working set of data fits into the cache, larger caches produce better performance; however, cache is expensive per byte compared to the disks. Therefore, system administrators must trade off between performance and cost.
- **Prefetching:** The prefetching length represents how much "extra" data the disk array reads and places in cache when a new read request is received. For example, if prefetching is set to 64KB, and a read request for location 100 misses in the cache, the disk array will read at least 64KB of data

beginning at location 100. The principle of spatial locality tells us that the application is likely to request the entire 64KB of data in the near future. By prefetching, the disk array will have this data waiting in the cache.

Increasing the prefetching length improves performance only to the extent that the prefetched data is eventually requested by the application. If the workload has a highly random access pattern, and the prefetched data is evicted from the cache without ever being requested, then the time spent reading the extra data is wasted, as is the space in cache occupied by the unused data.

- **High-water-mark:** Research has shown that a large percentage of data written to a disk is modified or deleted within 30 seconds. Thus, performance can be improved by holding the written data in cache until it is unlikely to be modified or deleted. Because it is not possible to know when data will next be modified, many disk arrays simply keep data from write requests in cache as long as possible — that is, until the cache gets full. When the write cache is full, some data (usually the least recently accessed) is evicted and written to disk.

  The high-water mark indicates how full the write-back cache should be before data is evicted and written to disk. High-water marks close to 1 allow data to remain in cache longer, thus increasing the chance the data will be modified and an unnecessary write avoided. However, if the cache gets so full that there is no room for new data, then those new requests will be delayed while old data is written to disk. This delay degrades performance. Thus, the high water mark must be set to balance between prematurely writing data to disk and allowing writes to become delayed.

When initially configuring a storage system, a system administrator must set each value properly. The prefetching length and high-water mark should be set to produce the best performance, whereas the amount of cache must be chosen to optimize some cost/performance metric. It is too time-consuming to evaluate the disk array for every possible configuration; therefore, some system administrators use a simulator (such as Pantheon) to estimate the effects of different configurations. Disk arrays are so complex that current simulators are not perfectly accurate. To account for this, system administrators use the simulators only to identify those configurations that are likely to be near optimal. They then take the time to evaluate the disk array given the most promising configurations.

I will evaluate whether the synthetic workloads specified by the Distiller can be used in place of actual workload traces as input to the Pantheon Disk array simulator. I will determine whether the system administrator will choose the same set of configurations based on the results of the synthetic workloads as she will if uses real traces instead. Specifically, I will do the following for each configuration option discussed above:

1. Take a workload trace, and use Pantheon to predict the mean response time for each reasonable configuration value (for example, 0, 1KB, 8KB, 16KB, 64KB, 128KB, and 256KB for prefetching length).
2. Repeat the above evaluations using a synthetic workload representative of the target workload.
3. Determine whether the best performance (e.g., lowest mean response times) occur for the same configuration values.

I will then evaluate the behavior of the disk array under the chosen configurations using both the workload trace and the synthetic workload and determine whether the synthetic workload leads the system administrator to choose the same configuration. (This, of course, assumes that I have access to the disk array modeled by Pantheon.)

Once a storage system is configured and in use, the system administrator must also occasionally adjust the configuration in response to changes in the workload. In this situation, it is much more practical to use a simulator to estimate the effects of a proposed change than to actually re-configure the storage system. As with the initial configuration situation, the results of the simulation need not predict the

mean response time with high accuracy; instead, the simulation need only indicate whether the proposed modification will significantly improve or degrade performance.

I will evaluate whether the synthetic workloads identified by the Distiller can be used in place of workload traces to estimate the effects of a proposed configuration change. Specifically, for each configuration decision discusses above I will:

1. Use Pantheon to estimate the mean response time of a workload trace under the current disk array configuration.
2. Use Pantheon to estimate the mean response time of a workload trace under the proposed new disk array configuration.
3. Use Pantheon to estimate the mean response time of a synthetic workload under the current disk array configuration.
4. Use Pantheon to estimate the mean response time of a synthetic workload under the proposed new disk array configuration.
5. Determine whether the changes in the predicted mean response time have the same order of magnitude.

Finding that synthetic workloads can be used in place of workload traces for configuration decisions may greatly simplify the process of configuring a disk array. The attributes on which the synthetic workloads are based may lead to simpler, faster disk array models that can more accurately evaluate configurations. Also, if the synthetic workloads are simple enough (i.e., based on a very few attributes), it may be easier to collect the information that serves as input to the simulator than to collect an entire trace. Similarly, if the synthetic workloads are simple enough, we can possibly design algorithms that monitor a disk array's workload and automatically adjust the configuration as appropriate. In other words, if the optimal prefetch length is determined by only a few parameters, then the disk array can easily monitor those parameters and adjust the prefetch rate as appropriate.

There are several reasons that the synthetic workloads generated by the Distiller may not work well. First, I believe that the attribute-values that must be maintained in a synthetic workload are determined by the disk array and its configuration. By changing the configuration, we may also change the set of attributes on which the synthetic workloads must be based. I expect that the configuration changes we are considering (cache, prefetching, and high-water-mark) will not have a large effect on the choice of attributes; however, should this set of experiments fail, it would suggest that the set of "important" attributes is much more dependent on the configuration than first thought.

Also, because current disk arrays are so complex, most current simulators make several simplifying assumptions. As a result, the set of attributes that determine the results of the simulator may be different than the set of attributes that determine the behavior of the disk array. I expect that the set of attributes for the simulator will simply be a subset of the set of attributes for the disk array; however, it is possible that the two sets are very different. In this case, we would learn that either the design of the Distiller is faulty, or that the simplifying assumptions on which the simulator is based are not as accurate as first believed.

6.2 Evaluation of synthetic workloads in place of ad-hoc modifications

One of the limitations of workload traces is that they do not support the evaluation of hypothetical situations. When designing and configuring a disk array, system administrators are interested in how the disk array will behave if the workload changes (for example, becomes faster, or more bursty, or more random). Unfortunately, a workload trace is not flexible. As a result, administrators must develop algorithms to modify workload traces in order to evaluate hypothetical situations. Synthetic workloads,

on the other hand, are defined by a set of attribute-values. We can make small changes to a synthetic workload by making small changes to the attribute-values.

One obvious ad-hoc modification to a workload trace is to increase its arrival rate by reducing the interarrival times in the trace by some constant factor. However, this reduces the length of the trace by the constant factor. Should the evaluation require a trace of the original length, the "faster" trace must be supplemented, either by replaying part of the trace, or with a synthetic workload.

I will evaluate whether using a representative synthetic results in a more accurate evaluation than simply reducing the interarrival times of a workload, then replaying part of the trace.

7 New generation techniques

This section discusses several possible new attributes and generation techniques for contribution 4. The library of attributes and corresponding generation techniques is currently too small to support a meaningful evaluation of the Distiller. Running the Distiller with only synthetic target workloads will be trivially successful; and running the Distiller with a production target workload will almost always result in hint instead of a representative synthetic workload. Consequently, I plan to develop several additional attributes and make several modifications to existing attributes.

Section 7.1 discusses several ideas for attributes and generation techniques that I may implement if necessary. Section 7.2 discusses some techniques for configuring attributes — specifically, choosing states for a Markov model. Section 7.3 explains how I will evaluate the new attributes.

7.1 Summary of potential attributes

I present here a few ideas and techniques that may contribute to a future I/O workload generation technique. I divide these into two groups: Those for which the corresponding attributes are well-defined, and those for which they are not.

7.1.1 Concrete ideas

The techniques presented here are very well defined. I know exactly which attribute-values I wish to reproduce. They have not been implemented for one of two reasons: Either I have not had time; or, I do not see an obvious algorithm which will reproduce the desired attribute-value.

Interreference temporal density: In [10], Conte and Hwu define the interreference temporal density function (IRTD) as a function f where $f(x)$ is "the probability of there being x unique references between successive references to the same item." In my case, an "item" is a disk sector. Conte and Hwu designed this function to be an intrinsic measure of locality — that is, a measure of locality independent of a particular system's design implementation.

The IRTD contains information that can be used to estimate a workload's cache hit ratio. For example, if we know that $\sum_{i=0}^{256MB} f(x) = .5$, then we can estimate that the cache hit ratio for an array with a 256MB LRU cache is about .5. This estimation may be very useful because I have access to neither the FC-60's cache replacement policy nor its prefetching algorithm. I hypothesize that that the cache replacement and prefetching algorithms cause the cache to behave somewhat like a larger or smaller LRU cache. For example, a workload that heavily utilizes the array's prefetching may behave somewhat as if it were run on an array with a 300MB LRU cache. Likewise, a workload that triggers prefetching, but does not use the prefetched data, may behave as if it has access to a smaller cache. (The unused prefetched information effectively wastes some of the cache space.)

I have not implemented IRTD as a generation technique because, although it is very simple to measure, it is not obvious how to generate a workload with a given IRTD. In fact, it may not be computationally feasible to generate a workload with a given IRTD.

Interreference spatial density: Conte and Hwu also present the interreference spatial density function (IRSD) [10]. For this function f, $f(x)$ is defined to be "the probability that between references to the same item, a reference to an item x units away occurs."

Proximity: Gomez and Santonja realized that the requests of several processes working concurrently can be interleaved in a way that masks the locality of each process [18]. Therefore, their access pattern generator mimics several processes writing in phases (called ON periods). (See the full description in section 2.3.)

I have developed a similar generator called Proximity. This generator attempts to reproduce attribute-values similar to Gomez and Santonja's access pattern generator, but without requiring that the trace contain a process ID. Proximity attempts to reproduce four attribute-values:

1. The percentage of sequential I/Os (i.e., the percentage of I/Os whose first block immediately follows the last block of the previous request). This measurement captures prefetching behavior.
2. The percentage of I/Os sequential to some recent I/O. (e.g., sequential to one of the previous 25 I/Os.) This measurement captures prefetching behavior in spite of interleaved requests.
3. The percentage of I/Os with a small jump distance (e.g., less than 64KB). This measurement captures both prefetching and short seek time behavior.
4. The percentage of I/Os with small jump distance relative to some recent I/O. This measurement captures both prefetching and short seek time behavior.

The proximity generator is not yet able to produce a workload in which all four values match those of the target workload; however, even with small errors, the resulting synthetic workload is reasonably representative.

7.1.2 Vague ideas

Most workload generation ideas from other research areas do not correspond directly to useful I/O workload generation techniques; however, these techniques usually contain ideas worthy of closer examination. Here I discuss a few of these ideas and how they may relate to the synthesis of I/O workloads.

Clustering: Many authors discuss how to use clustering and principle component analysis to generate a set of jobs with which an entire computer system can be evaluated [28, 13, 15, 14, 7, 8, 9]. In general, these techniques are successful because each job has approximately the same effect on the system each time is is executed. This assumption is not true for I/O workload requests. The first time a workload read request is executed, it is usually a cache miss. Subsequent requests for the same data are often cache hits. Also, the response time of a request also depends heavily on the location value of previous requests. The location of the previous request determines the seek time required to reach the new location (which could be zero, if the request has been prefetched). Consequently, many clustering and principal component analysis techniques cannot be directly applied to I/O workload generation.

Hong and Madhyastha, however, discovered that the behavior of a particular request depends upon only the most recent I/Os. In response, they developed a generation technique that applies clustering techniques to segments of the target workload. (See full description in section 2.3.) Their success suggests that other clustering techniques can also be adapted for use in I/O workload generation.

Processor Modeling: The evaluation of new processor designs requires the use of synthetic instruction sets. These sets are often generated based on the instruction mix of the target binary executable. Many papers discuss techniques of choosing representative instructions sets and basic blocks . Those papers that focus mainly on the memory access patterns (for example Conte and Hwu, discussed above) are most useful to us, because they also focus on locality of reference.

Texture Synthesis: Texture synthesis is the process of generating a large texture map based on a small sample. (A texture map is an image that appears to have some texture. For example, a picture of

drops of water.) I hypothesize that there are similarities between the patterns that represent texture in an image and the patterns that represent spatial locality in a workload trace. As a result, I believe that the texture synthesis technique of Wei and Levoy [34] can be modified and used to generate an access pattern. The challenge is that the obvious modification will result in a workload with too much temporal locality (i.e., the access pattern will re-use the same small set of location values). This challenge is the same challenge Hong and Madhyastha face in refining their clustering technique [21].

7.2 Configuring generators

This section discusses new techniques for automatically configuring attributes using parameters that are not found directly in the workload. In some cases, we can develop a new means of configuring an existing attribute instead of developing a new one. For example, when using a Markov model, the user must define the states Markov model's states. I discussed in section 2.3 the two obvious means of defining states; however, there are many other algorithms for choosing states.

Markov model: When using a Markov model generator, the user must specify a method of choosing states. I discussed in section 2.3 the two obvious means of defining states (based on percentages and percentiles). I present here a third method based on clustering.

The goal when using this method is to choose the desired number of states, then define the states (i.e., assign sectors to states) in a way that maximizes the number of self-loops (transitions that do not change state) while maintaining a roughly equal number of sectors per state (otherwise, the algorithm would simply put all the sectors into a single state). An exact solution is NP-hard, so I instead use the following approximation: I define a graph with one vertex per sector. The edges of the graph are weighted such that the weight on (x, y) represents the percentage of transitions from sector x that are to sector y. I then use METIS to partition the graph [1]. Each cluster becomes a state in the Markov model.

7.3 Evaluating new Techniques

To evaluate a new attribute and corresponding generation technique, I will use the same method described in section 3.2.3 to evaluate attributes. That is, I will generate a synthetic workload that uses the new attribute and compare the performance of that workload to a workload in which the corresponding parameter values are taken directly from the target workload. At this point I have not implemented or evaluated any new generators. I plan to wait and only implement those generators necessary to evaluate the Distiller.

8 Preliminary Results

This section presents the results that I have obtained this far. First, section 8.1 provides the details of the experimental environment. Section 8.2 presents in-depth measurements of the OpenMail workload.

8.1 Environment

8.1.1 Software environment

We use a number of software tools to implement our iterative method for summarizing a workload's important performance-related characteristics. First, using a trace collection tool, we generate an I/O trace of the application under study. We then use our workload characterization tool to calculate the attribute-values corresponding to the current attribute list. Next, we use the resulting workload characterization as input to our workload generation tool, which generates a synthetic workload with these

characteristics (attribute-values). We then issue that workload to a storage device and use our performance measurement tool to measure and compare the performance of the original and the synthetic workloads. In this section, we describe these tools and how we used them to implement our method.

Trace collection tool To collect the traces of real-world workloads, we use the Measurement Interface Daemon (midaemon) kernel measurement system. The midaemon is part of the standard HP-UX MeasureWare performance evaluation suite [23]. This daemon captures information about processor-, memory- and I/O-related events. The midaemon's I/O trace provides information about workload access patterns (e.g., interarrival time, operation type, request size and request location), as well as request completion latency.

Workload analysis tool Our workload analysis tool, Rubicon, takes a workload trace and an attribute list as input and produces a text file containing the characterization of the workload [31]. Rubicon is an extensible analysis tool that can perform a variety of analyses, ranging from the simplistic (e.g., average request size) to the complex (e.g., stack depth curve for request inter-reference distance). Its extensible structure means that experimenting with a new analysis technique is quite easy: one only needs to implement a new C++ analysis module. Rubicon has the ability to filter trace records, so that analyses can be applied to selected I/O trace records only (e.g., only reads or only accesses to a single disk array LU).

The attribute list, which is an input to Rubicon, indicates which analyses should be performed on which trace records (i.e., any filtering on the input trace fields), and how the results should be reported.

Performance measurement tool Rubicon also serves as our performance measurement tool, operating in offline mode on a trace to compute I/O request latency. (Recall that I/O request latency can be computed from the arrival and completion times collected in the midaemon's trace.) We simply implemented an analyzer module in Rubicon to compute the request latency CDF for a trace.

Workload generation tool Our workload generation tool, GenerateSRT, takes a workload characterization produced by Rubicon as input, and generates synthetic a workload file matching that characterization.[15] (Note that, at this point, the synthetically generated requests have not yet been issued to any device.) To synthesize a workload, GenerateSRT repeatedly generates requests by choosing a value for each of the four workload parameters. These values are chosen in one of several ways:

- randomly from an empirically determined histogram of values,
- randomly from a stochastic model (e.g., a Markov model), or
- deterministically from an explicit sequence of numbers, such as the list of location values as they appear in the baseline workload trace. (If the number generation for all parameters is configured in this manner, when the baseline and synthetic workloads are identical.)

Workload replay tool GenerateSRT generates a binary file that represents the synthetic workload. To determine the performance of this synthetic workload, we use a tool called *Buttress* to issue the I/Os to a storage device. While we run Buttress, we also run the midaemon trace collection tool to generate a trace of the synthetic workload for offline performance analysis by Rubicon.

Cache Flush tool This tool attempts to fill the cache with "dirty" data by repeatedly reading from random disk locations.

[15]The format of this file is known as *SRTlite*; hence the name "GenerateSRT"

8.1.2 Hardware environment

Unless specified otherwise, the experiments presented here were conducted on an HP FC-60 disk array, with workloads generated on a uniprocessor HP N4000 machine with a single 440 MHz processor and 1 GB of memory. The FC-60 array is populated with thirty 18 GB disks, spread uniformly across six disk enclosures, for a total of 0.5 TB of storage. The array has two redundant controllers in the same controller structure with one 40 MB/s Ultra SCSI connection between the controller enclosure and each of the six disk enclosures. The array is configured with five six-disk RAID5 LUs, each with a 16 KB stripe unit size.

The 256MB disk array cache, which is split between the two controllers, uses a write-back management policy backed with non-volatile RAM. Writes are considered to be completed once the data has been placed in the cache, resulting in a much lower latency than if they needed to be committed to the disk media. Thus, from the perspective of the user application, most writes will appear as cache hits (e.g., almost "free"), provided that the write portion of the cache isn't full. (We note that this condition doesn't occur for the workloads considered in this paper). See section 2.2 for more discussion of the FC-60 disk array.

8.1.3 Real Workloads

Here we give the high-level details of the real workload traces.

OpenMail The OpenMail traces are traces collected from an OpenMail e-mail server. The original workload trace was collected on an HP K-580 server system with an I/O subsystem comprised of four EMC Symmetrix 3700 disk arrays. The server was sized to support a maximum of about 4500 users, although only about 1400 users were actively accessing their e-mail during the trace collection period, which corresponded to the server's busiest hour of the day. The majority of accesses in the trace are to the 640 GB message store, which is striped uniformly across all of the arrays.

In order to create a baseline workload that has performance comparable to our synthetically generated workloads, we replayed a portion of the original trace corresponding to a single representative Symmetrix array on an FC-60 array. We use the performance information from that replay as our baseline in the presentation of the results (i.e. the "target" workload).

The high-level characteristics of our baseline Open Mail workload are as follows: The workload contains highly randomized accesses using small I/O requests that are mostly writes. Over 90% of the requests have request sizes of 8 KB or less, with about half of the requests being exactly 8 KB. The meta-data portion of the e-mail logical volume is frequently accessed, while the e-mail message (i.e, data) portion of the volume does not exhibit the same temporal locality. The complete workload is described in more detail in [22]. Our baseline trace contains 271,776 I/Os, with an average request rate of 75 requests per second, and an average throughput of 524 KB/s.

Cello The Cello workload is one hour of the traces collected on a workgroup file server at HP Labs in 1999. The workload contains highly randomized accesses using small I/O requests that are mostly (65%) writes. Most of the requests have a request size of 8KB or less, with those request sizes larger than 8KB being multiples of 8KB. Or target trace contains 54,175 I/Os with an average request rate of 30.4 requests per second.

8.2 OpenMail

This section presents the results of running the Distiller using the OpenMail workload. First we run the Distiller on each of the four LUs separately to see if the Distiller chooses different attributes for different LUs.

Workload	Attribute Set					
	Naïve	Best OpenMail	Best Synth1	Best Synth2	Best Synth3	Best Synth4
Open Mail Synth1 Synth2 Synth3 Synth4	.65 (.18)		.07 (.22)			

Table 8:

The following table presents the high-level characteristics of each LU

Workload	Num I/Os	% Read	Mean I/O Rate	Throughput	Mean error	(stdev)
Synth1	19,769	28%	22 io/s	164KB/s	??	??

Workload	Attributes Chosen
OpenMail	
Synth1	Distributions of request size, operation type, interarrival time, location of read request, location of write requests. Location values chosen by drawing values from distribution *without replacement.*
Synth2	

Table 8 shows the results. For these experiments, we used the FC-60 disk array, configured with five 85GB RAID 5 LUs. The segment size was 16KB. The target workload was 900 seconds long. In order to set the cache to a known state, the cache-flush program was run before each replay. In addition, we allow the disk array to sit idle for three minutes after running the cache-flush program. This gives the disk array time to finish writing back any data beyond the cache's high-water mark and settle into a steady state.

9 Conclusion

In summary, the Distiller will greatly improve our ability to generate representative synthetic I/O workloads. This improvement will lead to an improvement in our ability to evaluate disk arrays. The resulting improvement in performance prediction will allow us to more efficiently configure and allocate disk arrays. In addition, the additional understanding gained from observing how different characteristics affect performance will allow us to design better disk arrays. Furthermore, the Distiller will also ease world hunger, bring peace to the Middle East, make undergrads stop whining, help the Detroit Lions win the Super Bowl, and (if you ask really, really nicely) wash your car.

A Performance Comparison Algorithms

Here I present the pseudo-code for various algorithms calculating the difference the performance of two workloads. In each case, the performance of the workload is presented using a cumulative distribution function of latency. The CDF of latency is an increasing function $f : \mathbb{R} \rightarrow \mathbb{R}$ where $f(x)$ is defined to be the fraction of I/Os whose latency is at most x.

A.1 Root-mean-square

Given two cdfs of latency f_1 and f_2, the root-mean-square difference in performance 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)

A.2 Relative root-mean-square

Given two cdfs of latency f_1 and f_2, the relative root-mean-square difference in performance 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)

References

[1] Metis. Web page:http://www-users.cs.umn.edu/ karypis/metis/metis/index.html.

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

[3] E. Anderson. Simple table-based modeling of sorage devices. Technical Report HPL-SSP-2001-4, Hewlett-Packard Labratories, http://www.hpl.hp.com/SSP/papers/, 2001.

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

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

[6] R. R. Bodnarchuk and R. B. Bunt. A synthetic workload model for a distributed system file server. In *Proceedings of SIGMETRICS*, pages 50–59, 1991.

[7] M. Calzarossa and G. Serazzi. A characterization of the variation in time of workload arrival patterns. *IEEE Transactions on Computers*, C-34(2):156–162, February 1985.

[8] M. Calzarossa and G. Serazzi. Worklaod characterization: A survey. *Proceedings of the IEEE*, 81(8):1136–1150, August 1993.

[9] M. Calzarossa and G. Serazzi. Construction and use of multiclass workload models. *Performance Evaluation*, 19:341–352, 1994.

[10] T. M. Conte and W. mei W. Hwu. Benchmark chararacterization for experimental system evaluation. In *(The Hawaii Conference)*, 1990.

[11] M. Crovella and A. Bestavros. Self-similarity in world wide web traffic: Evience and possible causes. *IEEE/ACM Transactions on Networking*, 6(5):835–845, December 1997.

[12] M. R. Ebling and M. Satyanarayanan. SynRGen: an extensible file reference generator. In *Proceedings of SIGMETRICS*, pages 108–117. ACM, 1994.

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

[14] D. Ferrari. On the Foundations of Artifical Workload Design. In *Proceedings of SIGMETRICS*, pages 8–14, 1984.

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

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

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

[18] M. E. Gomez and V. Santonja. A new approach in the modeling and generation of synthetic disk workload. In *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of COmputer and Telecommunication Systems*, pages 199–206. IEEE, 2000.

[19] B. Hong and T. Madhyastha. The relevance of long-range dependence in disk traffic and implications for trace synthesis. In *Don't know yet.*, 2002.

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

[21] B. Hong, T. Madhyastha, and B. Zhang. Cluster-based input/output trace synthesis. Technical report, University of California at Santa Cruz, 2002.

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

[23] H. O. I. Lab. *HP OpenView Data Extraction and Reporting*. Hewlett-Packard Company, Available from http://managementsoftware.hp.com/library/papers/index.asp, version 1.02 edition, February 1999.

[24] A. Merchant and G. A. Alvarez. Disk array models in minerva. Technical Report HPL-2001-118, Hewlett-Packard Laboratories, 2001.

[25] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, march 1994.

[26] S. Sarvotham and K. Keeton. I/o workload characterization and synthesis using the multifractal wavelet model. Technical report, Hewlett-Packard Labs Storage Systems Department, 2002.

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

[28] K. Sreenivasan and A. J. Kleinman. On the construction of a representative synthetic workload. *Communications of the ACM*, 17(2):127–133, March 1974.

[29] M. Uysal, G. A. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. In *MASCOTS*, 2001.

[30] A. Varma and Q. Jacobson. Destage algorithms for disk arrays with nonvolatile caches. *IEEE Transactions on Computers*, 47(2), February 1998.

[31] A. Veitch, K. Keeton, D. Obal, and J. Wilkes. Rubicon. Technical report, HP Labs, 2001.

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

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

[34] L.-Y. Wei and M. Levoy. Fast texture synthesis using tree-structured vector quantization. In *SIGMETRICS 2000*, 2000.