Synthesizing Representative I/O Workloads Using Iterative Distillation (Extended Version — GIT-CERCS-03-29) Last revised: 18 December 2003

Zachary Kurmas
College of Computing
Georgia Tech
kurmasz@cc.gatech.edu

Kimberly Keeton
Storage Systems Department
Hewlett-Packard Laboratories
kkeeton@hpl.hp.com

Kenneth Mackenzie
College of Computing
Georgia Tech
kenmac@cc.gatech.edu

Abstract

Storage systems designers are still searching for better methods of obtaining representative I/O workloads to drive studies of I/O systems. Traces of production workloads are very accurate, but inflexible and difficult to obtain. (Privacy and performance concerns discourage most system administrators from collecting such traces and making them available to the public.) The use of synthetic workloads addresses these limitations; however, synthetic workloads are accurate only if they share certain key properties with the production workload on which they are based (e.g., mean request size, read percentage). Unfortunately, we do not know which properties are "key" for a given workload and storage system.

We have developed a tool, the Distiller*, that automatically identifies the key properties (more formally called attribute-values) of the workload. These attribute-values can then be used to generate a synthetic workload representative of the production workload. This paper presents the design and evaluation of the Distiller. We demonstrate how the Distiller finds representative synthetic workloads for simple artificial workloads and three production workload traces.*

1. Introduction

The behavior of most computer systems — especially large enterprise storage systems — is heavily dependent upon the choice of workload. Consequently, potential storage system design and configuration decisions must be evaluated with respect to workloads that represent how the storage system will be used in a production environment.

One approach for driving storage design studies is to use block-level storage traces from an actual storage system in production use. As described in [11], using traces of real storage system activity has a number of limitations: (1) Traces are difficult to obtain, often for non-technical reasons. System administrators are reluctant to permit tracing on production systems; and, when traces are collected, it is often time-consuming to anonymize them to protect users' privacy. (2) Although any single trace file may not be huge, a set of trace files listing the activity of a system over a longer period of time (weeks or months) may occupy considerable space (e.g., tens of GB), making them difficult to store online and share easily over the Internet. (3) It is difficult to isolate and/or modify specific workload characteristics (e.g., arrival rate or total accessed storage capacity) of a trace. This means that traces do not support hypothetical studies, such as explorations of slightly larger, busier, burstier, or other expected future workloads.

An alternative approach is to use synthetic workloads. Synthetic workloads are artificially generated workloads intended to induce similar behavior on the underlying storage system by preserving the properties of the target realistic workloads on which they are based (e.g., same request interarrival time distributions, request size distributions, operation mixes, and locality).

Synthetic workloads help to overcome many of the limitations of production traces: (1) Synthetic workloads can be specified using only the values of high-level attributes, which do not contain any user-specific information, thereby reducing privacy concerns. (2) Summarized workload attributes may be considerably smaller than a complete trace, making them easier to store and share over the Internet. (3) Synthetic workloads may better enable hypothetical studies: adjusting the attribute-values will change the resulting synthetic workload to approximate future workloads.

In order to be useful, a synthetic workload must be representative of the target workload on which it is based. In other words, it should induce similar behavior on the underlying storage system and lead to the same design decisions as the target workload [9]. Unfortunately, we do not know which attribute-values a synthetic workload must share with the target workload in order to be representative.

Most current workload analysis and synthesis techniques [11, 12, 13, 15, 16, 24, 25]. attempt to reproduce only one or two important workload properties. As a result, synthesizing a representative workload requires a tedious process of searching for each of the attributes that significantly affect storage system behavior. Furthermore, because the set of important attributes may differ for different workload/storage system combinations, this process must currently be repeated for every workload and storage system combination under study.

This paper presents our approach for automating this tedious process, and describes our tool, the Distiller. Beginning with a trace of the target workload and a set of candidate attributes, the Distiller automatically determines which attribute-values should be used to synthesize a workload that is representative of the target. For this paper, we consider two workloads to be representative when they have similar distributions of I/O response time when replayed on a given storage system.

We have used the Distiller to automatically find the key attributes for several simple workloads, and three production workloads — an email server, a transaction processing database, and a decision support database. We show that, for all but one of the target workloads considered, the Distiller produces a synthetic workload with a response time distribution within 12% of the target workload's response time distribution.

The Distiller chooses attributes from a library of known analysis and synthesis techniques. Should a necessary attribute be missing from the set, the Distiller can identify what I/O request parameters are measured by the missing attribute, thus helping to guide the invention of a new analysis/synthesis technique.

The Distiller can easily incorporate new attributes as they are discovered — either as part of this investigation, or from other investigations described in the literature. This extensibility allows the Distiller to easily evaluate new workloads and new storage technologies (e.g., MEMS-based storage), which may have different characteristics from known workloads and storage devices.

The remainder of this paper is organized as follows. Section 2 discusses related work, and Section 3 presents background and terminology. Section 4 discusses our automatic iterative approach for choosing attributes to generate representative synthetic workloads. Section 5 describes our experimental environment, and Section 6 presents our experimental results. Section 7 presents future work and section 8 concludes.

2. Related work

The literature describes and evaluates many techniques for generating synthetic block-level I/O workloads [11, 12, 13, 15, 16, 24, 25], file-level workloads [1, 6, 14, 22], and application-level I/Os workloads [18]. File- and application- level synthesis techniques are important because they can be used to produce block-level workloads. (Simply execute the synthetic file- or application-level workload, and collect the resulting block-level trace.) Furthermore, these I/O synthesis techniques are often based on, or related to, techniques for synthesizing other types of workloads, such as processor or network workloads [2, 3, 4, 5, 7, 8, 9, 10, 21].

Our contribution is different: instead of presenting another synthetic workload generation technique, the Distiller leverages these existing techniques to automatically choose the ones that are most appropriate for the target workload and storage system. The set of current block-level analysis and corresponding generation techniques serve as the Distiller's "library" of candidate attributes.

Researchers have used principle component analysis (PCA) to synthesize computational workloads (i.e., batches of jobs) [4]. PCA is applied to reduce the number of dimensions of a data set. Given a large set of workloads W with the desired performance and a set m of attribute-values that characterize those workloads, PCA finds basis vectors for the set of n-tuples that can best describe W. Unfortunately, using PCA to identify a workload's performance-related attribute-values presents several challenges. First, we would need to provide a set of workloads W with similar performance. (This, is in some sense, the problem we are attempting to solve.) Second, the resulting basis vectors would likely be combinations of attributes (rather than a subset of the initial set m), and may not have any intuitive meaning.

Techniques that use PCA and/or related clustering techniques to directly synthesize workloads must be designed carefully. Many PCA techniques (especially those used to synthesize computational workloads) assume that each component (job) consumes a fixed amount of resources (e.g., CPU time, I/O bandwidth, memory) each time it is issued. This assumption does not hold in a storage context, where the effects of spatial and temporal locality, caching, and prefetching can make the resources consumed for an I/O request highly variable. Thus, to use PCA and/or related clustering techniques to synthesize I/O workloads, one must define components to be something other than a single I/O request.

Hong and Madhyastha have applied clustering techniques to groups of I/O requests to produce a representative arrival pattern [15, 16]; however, this approach only covers a single aspect of the trace, without addressing the other as-

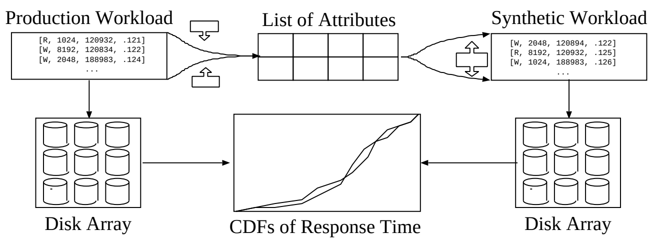

Figure 1. Problem statement: Our goal is to automatically determine which attributes are necessary to synthesize a I/O workload representative of the target.

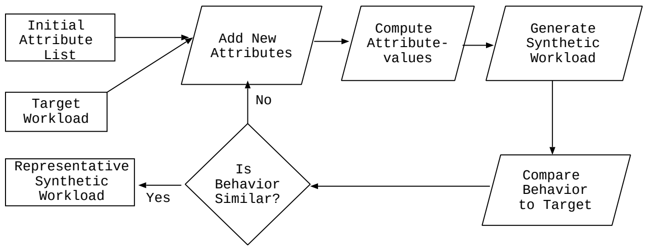

Figure 2. The Distiller iteratively builds a set of attributes that specifies a representative synthetic workload.

pects (e.g., access pattern). Wang, et al.'s PQRS method, which is based on the joint entropy of the location and arrival time values, provides a step toward capturing multiple trace characteristics simultaneously (e.g., arrival pattern burstiness, access pattern behavior and correlations between the two) [24]. These techniques serve as attributes in the Distiller's library (although, they were not needed to distill any workloads examined in this paper).

3. Terminology

Our goal is to automatically determine which attributes specify a synthetic workload that is representative of the target workload. Specifically, we want the synthetic and target workloads to have similar performance when played against the same storage system. Figure 1 illustrates this goal. In this section, we more precisely articulate this goal by defining terminology and evaluation metrics and by describing workload formats and target storage systems.

3.1 Storage system

We focus on block-level, disk-array-based storage systems commonly used in enterprise environments. Disk arrays provide a block-level I/O interface by exporting *logical units (LUs)* of storage. LUs are constructed from a subset of the array's disks and configured using a particular RAID layout (e.g., a RAID5 redundancy group). Each LU appears to be a single virtual "disk" to the host accessing it. Disk arrays also generally employ a large main memory cache to improve performance for frequently accessed data, leading to request response times that may vary by as much as three orders of magnitude.

3.2 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 1 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 1 also presents the jump distance and run length for the sample workload. These terms, defined in section 3.2.2, are used to more clearly define several workload attributes and generation techniques.

3.2.1 Open vs. closed model

The description of a workload presented in section 3.2 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.

Table 1. Example workload and closely related values.

	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

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 paper, we will use only the open model.

3.2.2 Descriptive terms

We define here several common terms that 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 1 form a run of length 3. I/Os 5 - 6 form a run of length 2.

Jump distance: The *jump distance* between two I/Os is the distance in the location address space from the end of one I/O to the beginning of the next. For example, in table 1, 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.

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 1 is

$$\begin{array}{ll} [1024, 18432) & [19456, 24576) \\ [33792, 41984) & [51200, 116736) \end{array}$$

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

Table 2. Distribution of request size for example workload

Request size	Number of I/Os	Fraction of I/Os
1024	2	.25
2048	1	.125
4096	1	.125
8192	3	.375
65536	1	.125

3.3 Attributes and attribute-values

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

Attributes describe a measurement of the workload itself, rather than the response of the underlying storage system when subjected to the workload. For example, "mean response time" is not a valid attribute because computing the mean response time requires knowledge of the specific storage system (and storage system configuration) on which the trace will be executed. Furthermore, attributes must be fully defined. For example, "locality" and "burstiness" are not valid attributes because there are many different ways to quantify locality and burstiness. In contrast, the Hurst parameter of interarrival times is a valid attribute.

The remainder of this section discusses several useful attributes.

Empirical distribution: We use the term *empirical distribution* to refer to the actual distribution of values for some I/O request parameter. For example, the example workload's distribution of request size is shown in table 2. In

Table 3. Sample transition matrix for operation type

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

contrast, many people will assume the parameter values follow some implicit distribution (e.g., a normal or Poisson distribution) and measure only the mean and standard distribution of the parameter in question.

In addition to measuring the distribution of values for a single request parameter, we can also measure the empirical distribution of tuples of I/O request parameters. For example, we can measure the distribution of (operation type, location) pairs. Such distributions are often called *joint distributions*.

Conditional distribution: An empirical distribution measures a request parameter's distribution of values over all of a workload's requests. We can instead measure the distribution of values for only those I/Os that meet a specified condition. For example, we can calculate separate distributions of location for read requests and write requests. Likewise, we can partition the range of locations into 10 states, then calculate separate distributions of location values for each state. We call the parameter being measured the *dependent parameter*; the parameter on which the condition is based is the *independent parameter*. In the first case, location is the dependent parameter, and operation type is the independent parameter. In the second case, location is both the dependent and independent parameter. As with empirical distributions, the distributions measured can be joint distributions. Similarly, the independent parameter may a tuple of parameter values.

State transition matrix: A state transition matrix takes a partition of an I/O parameter's range into states and, for each pair of states (x, y), lists the probability that an I/O request taking on a value corresponding to state x is followed by a request taking on a value corresponding to state y. The Table 3 shows that the probability an I/O request is a read is 75% if the previous request was a read, and 20% if the previous request was a write.

The states for a transition matrix can be defined in any arbitrary manner. One obvious technique is to assign each value in the request parameter's range to a unique state. This works well for operation type because there are only two states (read and write); however, because the size of the matrix is quadratic in the number of states, this technique is rarely used to study the other request parameters.

For our research, we usually divide a parameter's range into states according to percentiles. For example, we partition the range of location values into 4 states as follows:

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

This method produces a set of states for which there are an equal number of I/Os corresponding to each state. After testing several different techniques, we have found this one to be most useful in practice.

The states can also be defined to be tuples of parameter values. These tuples may be values for different parameters of the same I/O request (e.g., (operation type, location)), or values from successive I/O requests (e.g., (previous location value, current location value)). When specifying a transition matrix based on tuples, one must be mindful of the total number of states, as it grows exponentially with the dimension of the tuple.

Jump distance: This is simply the empirically observed distribution of jump distances between adjacent I/Os. Jump distance can serve as either a dependent or independent parameter for both transition matrices and conditional distributions.

Run count: This is simply the empirically observed distribution of run lengths. Notice a workload may contain fewer runs than I/Os because each run comprises several I/Os. As with jump distance, run count can serve as either a dependent or independent parameter for both transition matrices and conditional distributions.

Jump distance within state: The jump distance analyzer calculates jump distance as the difference between the beginning of the current request and the end of the previous request. Jump distance within state calculates the jump distance between the beginning of the current request, and the end of the most recent request corresponding to the same state as the current request. For example, consider the following sequence of locations: 10, 11, 12, 20, 21, 13, 22, 14, 23, 24, 15. Assume each request size is 1 unit, and define states $[10, 19]$ and $[20, 29]$. The jump distance within state for location 22 would be 0, because the previous location in state $[20, 29]$ is 21.

Run count within state: The run count analyzer considers only runs of strictly sequential I/Os. The run count within state analyzer looks to the previous location in the same state to determine the length of a run. In the previous example, this analyzer would find two runs: 10, 11, 12, 13, 14, 15; and 20, 21, 22, 23, 24, 25. Notice that for run count within state to work as intended, the runs must lie within different states.

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

1. the *dependent parameter, d*: the request parameter(s)[1] being measured
2. the *independent parameter, i*: the request parameter(s)[1] on which the states are based;
3. the *number of states, s,* used to express the independent parameter; and
4. the *history h*: the number of previous I/Os considered

This template specifies a set of s states based on the h most recent independent parameters[2] . We then generate a conditional distribution of the dependent parameter based on the defined states. We can also optionally generate the Markov transition matrix for the same set of states. Together the conditional distribution and Markov transition matrix form a Markov model. (Notice that when $h > 1$, each Markov state does not correspond directly to a single I/O request.)

3.4 Evaluation criteria

Evaluation criteria: We can determine behavioral similarity by considering a variety of different behaviors (response time, throughput, power consumption, etc.). The similarity of each behavior can be quantified using many different metrics. For example, we can compare distributions of response time using root mean square distance or the Kolmogorov-Smirnov test. The design of the Distiller is independent of the performance metric and similarity measure chosen.

In this paper, the disk array behavior in question is the response time distribution: the Distiller's synthetic workload should maintain the same response time distribution as the target workload when both are played against the same storage system. Our similarity metric is the *demerit figure* [20]. The demerit figure is the root mean square of the horizontal distance between the response time cumulative distribution functions (CDFs) for the synthetic and target workloads. We will present the demerit figure in relative terms, as a percentage of the mean response time of the target workload.

A synthetic workload that perfectly represents the target workload trace has a demerit figure of 0%. However, due to various experimental errors, it is difficult to achieve identical performance. Ganger distinguishes between *synthesis error*, due to the different synthesis techniques, and *randomness error*, the error of a single synthesis technique using different random seeds [11]. Because we are playing requests against a real storage environment, we may also experience *replay error*: the experimental error due to non-determinism in the disk array and host operating system. Our experiments indicate that replay error can be as high as 10%; we therefore set our target at 12%, allowing for an additional 2% synthesis error and randomness error.

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

[2]In practice, we use only the aforementioned "percentile" method; however, the method of defining s states could be viewed as fifth parameter to this template

4. Our approach

In this section, we present our iterative approach for determining which attributes are necessary for synthesizing a representative I/O workload. This approach is embodied in a tool we call the *Distiller*.

At a high level, the Distiller iteratively builds a list of "key" attributes. During each iteration, the Distiller identifies one additional key attribute, adds it to the list, then tests the representativeness of the synthetic workload specified by the current list of key attributes. This loop (shown in Figure 2) continues until either (1) the difference between the performance 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.

Running Email example: To make the concepts more concrete, we will use a running example to illustrate the Distiller's operation on a real production workload. Progress will be described in each section, and the results summarized in Table 8.

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

4.1 Initial attribute list

The Distiller's first step is to generate a synthetic workload based on a set of empirical distributions of values for the four I/O request parameters. We start with these explicit distributions because implicit distributions (e.g., normal or Poisson) have been shown to be inaccurate [11].

Running Email example: Figure 3 shows the response time distributions for the initial synthetic workload and the target workload, which result in a demerit figure of 65%. Note the log scale on the x-axis. Given that the demerit

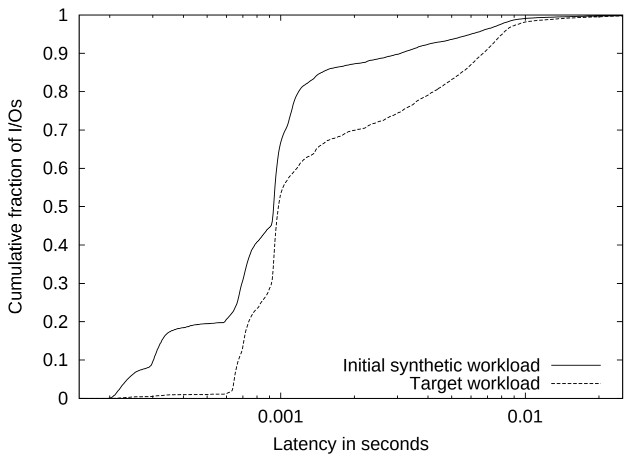

Figure 3. The Distiller cannot accurately synthesize the target Email workload using only the initial attribute list of empirical distributions for the trace parameters.

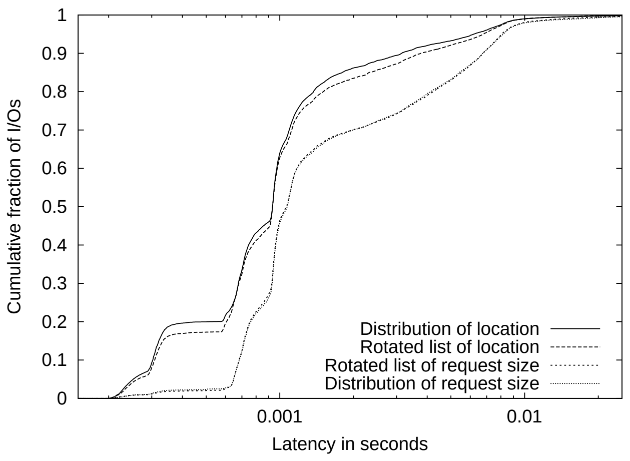

Figure 4. The difference between location lines (15% demerit) indicates the need for a $\{$location$\}$ attribute. The similarity of the request size lines (8% demerit) indicates no need for a $\{$request size$\}$ attribute.

figure is larger than the threshold of 12%, the Distiller must search for additional attributes.

4.2 Choosing an attribute group

Because the time to evaluate a synthetic workload's response time distribution is proportional to the length of the target workload, the Distiller should evaluate attributes in an intelligent order. Instead of evaluating attributes individually in an arbitrary order, the Distiller estimates the maximum potential benefit of an entire group of related attributes. Each group of related attributes captures a particular relationship within a single workload parameter or between multiple parameters. Once the Distiller determines which group holds the most potential for improvement, it can focus on finding the appropriate attribute to capture that relationship.

We define an *attribute group* as a set of attributes whose values are calculated using the same set of I/O request parameters (and thus capture information about the same relationship). For example, the $\{$interarrival time$\}$ attribute group contains those attributes that measure only the interarrival times of different requests (e.g., mean interarrival time, the Hurst parameter). The $\{$location, operation type$\}$ attribute group contains those attributes that measure the relationship between requests' locations and operation types (e.g., separate distributions of location values for read and write requests). All possible combinations of parameter relationships result in a total of fifteen attribute groups. By definition, each attribute is a member of exactly one attribute group.

To evaluate the potential benefit of an attribute group, the Distiller examines what happens when the relationship captured by the attribute group is destroyed. If a synthetic workload without the relationship under test performs similarly to the target workload, then the attribute group (and hence all of its member attributes) provides little or no benefit. However, if performance without the relationship is dramatically different, then the attribute group is important. We call this approach to evaluating the importance of attribute groups the *subtractive* method. The goal is to isolate the contribution of the relationship represented by the attribute group under test. For example, to isolate the effects of the $\{$request size$\}$ attribute group, we want to separate its contribution from the contributions of $\{$request size, operation type$\}$, $\{$request size, location$\}$, and $\{$request size, interarrival time$\}$, as well as all three- and four-parameter attribute groups involving request size.

We isolate a given attribute group by replacing its parameter values in the target trace with either an empirical distribution or a rotation of the original values. The Distiller evaluates the potential contribution of a single-parameter attribute group using both of these techniques. Table 4 provides an example application of the subtractive method for $\{$request size$\}$. The left panel of the table shows the target workload. The second panel (labeled "Empirical Request Size") substitutes an empirical distribution for the request size parameter list, effectively removing all relationships from any of the groups involving request size ($\{$request size$\}$, $\{$request size, location$\}$, etc.). The third panel (labeled "Rotated Request Size") rotates the list of request sizes to maintain the intra-request size relationships, while destroying relationships between request size and the other parameters.[3] The attribute group's contribution can then be

[3]To rotate a list L of values, we shift the list so that the order of the values is unchanged, but the item in position x is now in position $(x +$

Table 4. Examples of the subtractive method, using empirical distribution substitution and list rotations.

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

isolated by comparing the difference between the response time distributions (using the demerit figure) for the empirical and the rotated workloads.

The Distiller evaluates the potential contribution of a multi-parameter attribute group using a combination of list rotations. First, for a parameter p, it evaluates the potential of all $\{p,\, x\}$ relationships (where x is another parameter) by comparing the performance for a rotated p list to the target workload. If the performance is sufficiently different, then at least one of these multi-parameter relationships must be important. The Distiller then isolates the contributions of each potential $\{p,\, x\}$ pairing using two synthetic workloads with different rotations. Table 4 provides an example application of the subtractive method for {request size, operation type}. Again, the left panel represents the target workload. The fourth panel (labeled "Rotated Together") shows how request size and operation type can be rotated together, preserving the {request size, operation type}, {request size} and {operation type} relationships, while destroying the other relationships involving request size and operation type. The right panel (labeled "Rotated Apart") shows how separately rotating the two lists further destroys the {request size, operation type} relationship. The attribute group's contribution can then be isolated by computing the difference between the response time distributions (i.e., the demerit figure) for the "rotated together" workload and "rotated apart" workloads.

The Distiller's next step depends on the result of the demerit calculation for the isolated attribute group. If the demerit figure is less than the user-defined threshold, then the potential contribution for the attribute group is small enough that the addition of further attributes from that group is unwarranted. Otherwise, the attribute group's potential contribution is large enough that the Distiller should determine which attribute from the group to add to the list of key attributes.

In exploring attribute groups, the Distiller first evaluates the importance of single-parameter attribute groups (e.g., {location}, {request size}, {operation type}, and {interarrival time}). We refer to these iterations as *phase one* of the algorithm. After examining the single-parameter attribute groups, the Distiller addresses two-parameter attribute groups in *phase two*. Finally, if necessary, the Distiller will search for important three-parameter and four-parameter attributes. However, we have yet to encounter any workloads for which it is necessary to proceed beyond phase two.

Running Email example: The Distiller begins its exploration of attribute groups by applying the subtractive method to each single-parameter attribute group. Figure 4 shows two representative results for {request size} and {location}. The empirical distribution and rotated workloads for {request size} are similar, with a demerit figure of only 8%; thus, no additional {request size} attributes (beyond the default empirical distribution) are necessary. Although the distributions for the two {location} workloads look similar, the demerit figure of 15% is above our threshold of 12%. Therefore, the Distiller will search for more informative {location} attributes.

4.3 Picking an attribute

Once the Distiller has identified a promising attribute group, it must choose a specific attribute from that group. It explores the candidate attributes in a pre-determined order and evaluates how close each attribute comes to achieving the potential contribution of the attribute group. The Distiller incorporates the first eligible attribute encountered. This first-fit criterion allows us to avoid the execution time overhead of extra attribute evaluations.

The Distiller uses a variant of the subtractive method to evaluate candidate attributes. It generates a synthetic workload by using the candidate attribute, and preserving the list of original values for parameters not associated with the attribute under test. The Distiller then compares this synthetic workload against the synthetic workload that main-

$t) \bmod length(L)$ for some constant integer t.

Table 5. Candidate attributes.

Attribute	Attr. Group	Description
empirical distribution	Any	histogram of values for this parameter obtained from the workload trace (This is the initial attribute for each parameter.)
list of values	Any	observed list of of values for the parameter in the target workload trace (This is the "perfect" attribute; it is used only for evaluation/comparison purposes)
Markov model	Any	higher-order Markov model with n different states, corresponding to different regions of the distribution; transition probabilities are determined empirically.
jump distance	{loc, req. size}	histogram of jump distances (in KB) in workload.
modified jump distance	{location}	Like jump distance, except jump distance is calculated from the beginning of the previous request to the beginning of the current request.[4]
run count	{loc, req. size}	Histogram of the length of runs observed in workload.
modified run count	{location}	Like run count, except runs determined using only location at the beginning of requests.
Markov model using combined attributes	{location}	Markov model using jump distance or run count.

tains "perfect information" for the attribute group (the "rotated" workload for single-parameter attributes, and the "rotated together" workload for multi-parameter attributes). If the two workloads have similar behavior (e.g., a demerit figure within the given threshold), the Distiller adds the attribute to the list of key attributes. If the two workloads have very different behavior, the attribute is not helpful and the Distiller proceeds to evaluate other candidate attributes.

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

4.4 Attribute library

The Distiller implements a library of attributes described in the research literature. Table 5 describes the attributes for which we have implemented analysis and synthesis techniques.

Running Email example: Recall from our earlier example that the Distiller had identified the {location} attribute group as the most promising. To explore this group, the Distiller first evaluates a Markov model of locations — MM(loc, loc, 100,1). Figure 5 shows that the Markov model-generated synthetic workload behaves very much like a workload with the original, rotated sequence of location values. Therefore, the Distiller adds the Markov model of locations to its key attribute list.

[4]The literature traditionally defines jump distance as the distance between the end of the previous request and the beginning of the current request. Our modified definition allows jump distance to be strictly a {location} attribute.

4.5 Tracking progress for each iteration

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

Running Email example: Figure 6 shows the results for the improved attribute list containing the Markov model of location values. Because the demerit figure (54%) is still well above the desired threshold, the Distiller continues.

4.6 Subsequent phases of Email example

After addressing each single-parameter attribute group, the Distiller addresses the two-parameter attribute groups. Recall that the Distiller begins this phase by comparing the single-parameter rotated workload for each I/O parameter p to the original workload trace to evaluate the potential of all $\{p, x\}$ multi-parameter attributes (where x is any other I/O parameter). It then determines whether breaking these multi-parameter relationships has a large effect on the resulting synthetic workload's behavior.

Figure 7 illustrates this process. We see little difference in behavior when {request size, w} and {interarrival time, x} relationships are broken; both demerit figures are less than 5%. However, the behavior of the rotated workloads for operation type and location differ significantly (demerit figures of 50% to 60%) from that of the target workload. Therefore, we conclude that some {operation type, y} attribute and some {location, z} attribute have a significant effect on behavior. The Distiller next identifies appropriate values for y and z by comparing the "rotated together" and the "rotated apart" workloads, as described in Section 4.2.

In the case of operation type, the Distiller evaluates

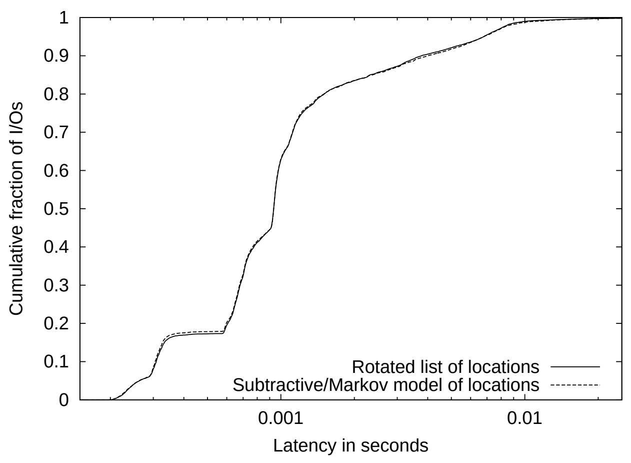

Figure 5. Markov model-generated location values are representative of the target workload's location values, so the Distiller adds the attributes to the list of key attributes.

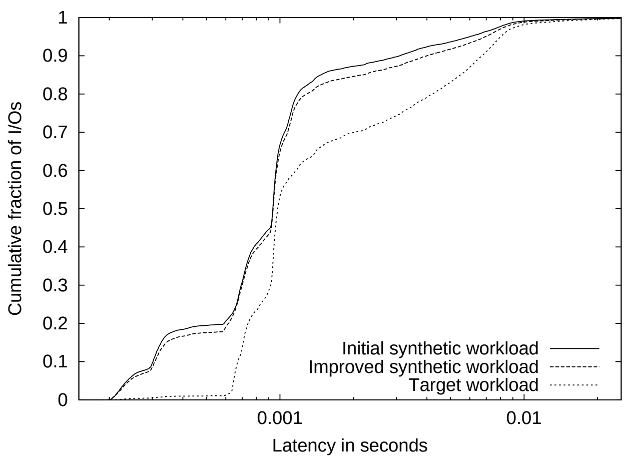

Figure 6. Although using a Markov model to choose location values improves the accuracy of the resulting synthetic workload, the improved workload is still not sufficiently representative.

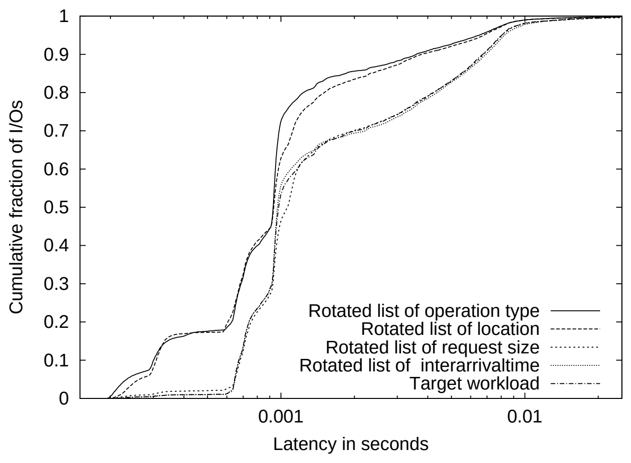

Figure 7. Inter-parameter relationships are important for operation type and location, but not for request size and interarrival time.

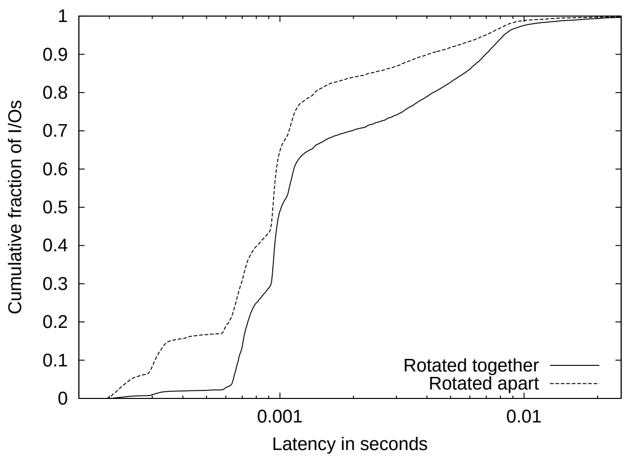

Figure 8. The potential for $\{$operation type, location$\}$ is high, indicating that some attribute in this group will significantly affect performance.

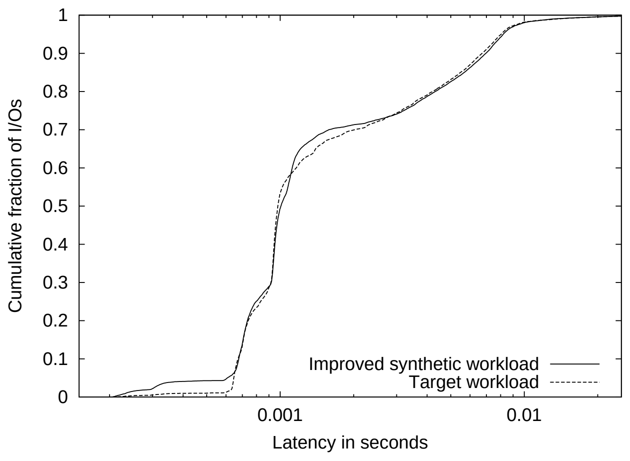

Figure 9. The improved synthetic workload (using separate Markov models for reads and writes) is representative of the target Email workload.

the potential contribution of the {operation type, location}, {operation type, request size}, and {operation type, interarrival time} attribute groups. The high demerit figure (56%) in Figure 8 indicates that there is an important {operation type, location} attribute. Other experiments show that the {operation type, request size} and {operation type, interarrival time} attribute groups promise little benefit.

When the Distiller has identified an important two-parameter attribute group, it evaluates the candidate attributes as described in Section 4.3. For our example, the Distiller first evaluates separate Markov models of location values for read requests and write requests. The resulting synthetic workload has similar behavior to the workload where operation type and location were rotated together. Therefore, the Distiller adds this attribute to the list of key attributes.

Figure 9 shows the results for the Distiller's evaluation of the synthetic workload specified by the improved attribute list. Because the demerit figure (8%) is below the desired threshold, the Distiller terminates.

5. Experimental environment

In this section, we describe our experimental software and hardware environment.

5.1 Software environment

The Distiller is responsible for applying the various subtractive methods, running the resulting experiments, examining the results, choosing the attribute group for improvement, and determining which attributes should be added to the list of important attributes. It acts as an outer loop that coordinates the activities of several other software tools that help perform these tasks.

To collect the traces described in Section 3, we use the Measurement Interface Daemon (midaemon) kernel measurement system, part of the standard HP-UX MeasureWare performance evaluation suite [17]. The midaemon's I/O trace provides information shown in table 6. From this data, we can extract the four parameters necessary for the workload traces (i.e., interarrival time, operation type, request size and request location), as well as the information necessary to calculate the response time of individual I/O requests (completion time and arrival time).

Our workload analysis tool, Rubicon, takes a workload trace and an attribute list as input and produces a text file containing the attribute-values that characterize the workload [23]. Rubicon also serves as our performance measurement tool, operating in off-line mode on a trace to compute the distribution of I/O request latencies.

Our workload generation tool takes a workload characterization produced by Rubicon as input, and generates a file containing a synthetic workload matching that characterization. (See appendix A for more details.) We then use a tool called Buttress to issue the requests to a storage device. While we are running Buttress, we also run the midaemon trace collection tool to generate a trace of the synthetic workload for off-line performance analysis by Rubicon.

5.2 Hardware environment

All of the experiments presented in this paper 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 complete once the data has been placed in the cache, and then later committed to the disk media (a process called "destaging"). Normally, destaging occurs as a background task; if the write portion of the cache fills, writes may be destaged in the foreground, delaying the completion of further requests. 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 is not full.

6. Experimental results

In this section, we present the results of using the Distiller to produce representative synthetic workloads for sev-

Table 6. midaemon I/O trace fields.

Field	Description
arrival time	the time at which the request arrived at the device driver
start time	the time at which the request was issued to the I/O device
completion time	the time at which the request was completed
operation type	whether the request is a read or a write
device identifi er	the storage device being accessed by this request. In a disk-array-based storage system, the device ID is an LU id.
address	the location on the storage device accessed by this request
request size	how many bytes of data are read or written

eral target workloads. We examine both artificial workloads and production workloads. Artificial workloads are simple workloads generated from the Distiller's library, intended to verify that the Distiller works correctly. Production workloads are workloads collected on real production enterprise storage systems.

6.1 Artificial workloads

The Distiller will be successful only if its library contains a sufficiently broad selection of attributes. Regardless of the breadth of the library, we want to verify the correct operation of the Distiller's infrastructure. To do this, we generated a set artificial workloads based on attributes in the Distiller's library. Because we know the Distiller's library contains all necessary attributes for the artificial workloads, any failure to produce a representative synthetic workload will indicate a bug or design flaw in the Distiller, not a a limitation of the library.

Table 8 presents the results of applying the Distiller to the artificial workloads in Table 7. Unless otherwise noted, the stopping condition for each workload is a demerit figure of 12%.

We now briefly describe each workload and the features of the Distiller tested by that workload. (In the list that follows, the number corresponds to the workload ID shown in Tables 7 and 8.)

W1 and W2 are simple workloads that are completely described by the empirical distribution attributes on the initial attribute list. The Distiller stops before even entering iteration 1 in both cases.

W3 demonstrates the Distiller's ability to find single-parameter attributes. The Markov models produce dependencies within the sequence of request parameters, but no inter-parameter dependencies.[5]

W4 shows that the Distiller can handle the temporary degradation in the accuracy of the synthetic workload when the addition of an important attribute does not improve the representativeness of the resulting synthetic workload.

When distilling a workload, the addition of an important attribute does not necessarily produce a more representative synthetic workload. Two (or more) important attributes may have offsetting effects. For example, a $\{$location$\}$ attribute may add sequentiality and speed up the resulting synthetic workload, while an $\{$interarrival time$\}$ attribute added in a later iteration may cause burstiness that slows the resulting synthetic workload. Consequently, the improvement in the synthetic workload may not become apparent until after both attributes have been added.

W5 demonstrates that the Distiller correctly chooses a useful attribute for the chosen attribute group, without settling on the first attribute it evaluates. During iteration 1, the Distiller tests and rejects three Markov models of location, and, instead, chooses a Markov model of jump distance.

W6 highlights the Distiller's ability to find attributes that describe multi-parameter correlations. In this workload, read and write accesses are concentrated in different areas of the LU's address space, and have different request sizes. In addition, successive reads and successive writes have smaller interarrival times than a read followed by a write, or a write followed by a read. The Distiller correctly skips over single-parameter attribute groups and finds the appropriate multi-parameter attributes.

W7 shows that the Distiller can find a useful set of attributes, even if no attribute corresponds directly to the generation techniques. Run count in this workload varies uniformly from 1 to 10 requests. (These runs were generated using a special run count generator, not a Markov model.) Even though simple Markov models can only generate runs with an exponential distribution, such a Markov model is sufficient to specify a representative workload. Thus, the Distiller shows us that capturing runs is important, but that maintaining the exact run length is not (at least for this workload).

The evaluation of these artificial workloads highlights two of the Distiller's strengths. First, the Distiller properly chooses attributes that produce representative synthetic workloads for the artificial workloads examined. (We have

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

Table 7. Workload parameters for target artificial workloads.

ID	Location	Operation Type	Interarrival Time	Request Size
W1	Uniform(0, 9GB)	50% reads	Constant(20ms)	Constant(8KB)
W2	Uniform(0, 9GB)	33% reads	Exponential(20ms)	Uniform(1KB, 128KB)
W3	MM(loc, loc, 4, 1) each state 96MB	MM(op, op, 2, 1)	MM(iat, iat, 3, 1) [.2ms, .9ms],[1ms, 5ms],250ms	MM(size, size, 4, 1) 1KB, 16KB, 64KB, 128KB
W4	MM(loc, loc, 4, 1) each state 96MB	MM(op, op, 2, 1)	MM(iat, iat, 3, 1) [.2ms, .9ms],[1ms, 5ms],250ms	MM(size, size, 4, 1) 1KB, 16KB, 64KB, 128KB
W5	MM(jump distance, loc, 4, 1) 98% probability of jump length determined by location	MM(op, op, 2, 1)	MM(iat, iat, 3, 1) [.2ms, .9ms],[1ms, 5ms],250ms	MM(size, size, 4, 1) 1KB, 16KB, 64KB, 128KB
W6	MM(loc, op, 2, 1) [0, 64MB] 95% R, 5% W [65MB, 10GB] 5% R, 95% W	MM(op, op, 2, 1)	MM(iat, op, 2, 2) W, W: .6ms R, W: 100ms W, R: 25ms R, R: 6ms	MM(iat, op, 2, 2) W, W: 128KB R, W: 65KB W, R: 2KB R, R: 16KB
W7	Runs of length Uniform(0,10) [0, 64MB] 90% R, 10% W [65MB,10GB] 10% R, 90% W	MM(op, op, 2, 1)	Four threads, each Exponential with following means: .03ms, .04ms, .05ms, .035ms	Constant(8KB)

Table 8. Summary of selected workload results.

ID	Iter.	Attr. group	Attribute added	Result
W1	0		empirical distributions	3%
W2	0		empirical distributions	6%
W3	0		empirical distributions	60%
	1	{loc}	MM(loc, loc, 100, 1)	66%
	2	{op}	MM(op, op, 2, 8)	42%
	3	{size}	MM(size, size, 100, 1)	9%
	4	{iat}	MM(iat, iat, 4, 3)	5%
W4	0		empirical distributions	15%
	1	{loc}	MM(loc, loc, 10, 2)	22%
	2	{size}	MM(size, size, 100, 1)	9%
Email	0		empirical distributions	65%
	1	{loc}	MM(loc, loc, 100, 1)	56%
	2	{op, loc}	MM(loc, op, 2, 1)	6%
OLTP Log	0		empirical distributions	29%
	1	{loc}	MM(jump dist., loc, 100, 1)	6%

ID	Iter.	Attr. group	Attribute added	Result
W5	0		empirical distributions	63%
	1	{loc}	MM(jump dist, loc, 100, 1)	23%
	2	{size}	MM(size, size, 100, 1)	13%
	3	{iat}	MM(iat, iat, 100, 1)	11%
W6	0		empirical distributions	87%
	1	{op,size}	MM(size, op, 2, 1)	54%
	2	{op, loc}	MM(loc, op, 2, 1)	27%
	3	{op, iat}	MM((op,iat), (op, iat), 8, 2)	5%
W7	0		empirical distributions	78%
	1	{loc}	MM(jump dist, loc, 100, 1)	74%
	2	{op}	MM(op, op, 2, 8)	30%
	3	{op, loc}	MM(jump dist, (op, loc), 100, 1)	7%
OLTP LU 1	0		empirical distributions	53%
	1	{loc}	MM(loc, loc, 100, 1)	34%
	2	{op, loc}	MM(jump dist, op, 100, 2)	13%
DSS	0		empirical distributions	68%
	1	{loc}	Run Count Within State	18%

also demonstrated this correctness for many other artificial workloads, not presented here due to space considerations.) Second, the Distiller was able to identify which attributes were important, regardless of the techniques that actually generated the target artificial workloads.

6.2 Production workloads

In this section, we apply the Distiller to production workloads. Table 8 presents a summary of the results.

Email: Section 4 presented a detailed description of the Distiller's operation on one LU of this workload. It chooses a Markov model for location, which is later subsumed by separate Markov models for location based on operation type (i.e., {operation type, location}) to generate a final synthetic workload with a demerit figure of 6%.

OLTP log: This trace was collected in 1994 while running HP's Client/Server TPC-C-like online transaction processing (OLTP) application at approximately 1150 transactions per minute on a 100-warehouse database. This workload focuses on the busiest LU, where accesses are highly sequential and write-only. The average request rate is 90 I/Os per second with an average throughput of 473KB/s. The Distiller completes this trace in only one iteration, as only an improved {location} attribute proves to be necessary.

OLTP LU1: Accesses to the second-busiest LU in this workload are about 70% reads, with an average request rate of 90 I/Os per second and an average throughput of 57KB/s. A visual inspection of the target trace shows that the access pattern tends to have groups of I/Os with similar addresses, but no obvious pattern (e.g., sequential runs or strides) within each group. This workload is distilled in two iterations, using a Markov model of location, which is later subsumed by Markov model of jump distance as a function of operation type, resulting in a final demerit figure of 13%.

DSS: This decision support system (DSS) trace was collected on an audited TPC-H system running the throughput test (multiple simultaneous queries) on a 300 GB SF data set. Accesses to this LU are read-only and nearly all requests are 128 KB. The average request rate is 50 I/Os per second, with an average throughput of about 6400 KB/second. Each query generates a sequence of sequential I/Os. Visual inspection of the trace shows that many independent sequential streams have been interleaved together. This pattern does not match any of the previously described attributes in the Distiller's library.

When given this trace, the Distiller first identifies the need for a better {location} attribute. It then evaluates every {location} attribute in the library and finds (as shown in Figure 10) that they all have demerit figures of at least 80%. Thus, the Distiller reports that the selection of {location} attributes is insufficient and terminates.

In response, we added an analyzer called "run count within state". Instead of measuring jump distance from the previous I/O (recall that a run is a sequence of I/Os with a jump distance equal to the request size of the previous request) this analyzer measures jump distance from the previous *nearby* I/O. (Here "nearby" refers to spatial locality). Thus, this analyzer can capture sequential runs, even if several runs are interleaved (provided that the interleaved runs are in different areas of the LU's address space). We reran the Distiller after adding the new analyzer to its library. The demerit figure obtained from evaluating the potential of "run count within state" is only 17%, a great improvement over the existing set of attributes. The original an final synthetic workloads are shown in Figure 11.

The DSS workload illustrates the Distiller's ability to help direct the development of new attributes when necessary. Because of the Distiller's extensible structure, it is easy to add analysis and synthesis modules for a new candidate attribute.

7. Ongoing and future work

In this paper, we have emphasized the ability of the Distiller to identify attributes necessary to generate representative synthetic workloads; however, this is just one piece of our research into identifying and understanding a workload's key attributes. By better understanding how a workload's attributes affect disk array behavior, we hope to improve our ability to obtain workloads for evaluation studies (especially synthetic workloads representative of hypothetical future conditions). Furthermore, we hope that this knowledge will aid the design of storage system hardware, firmware, configuration policies, and analytic models. For example, by learning precisely which attributes have the largest effect on the performance, we may learn how to precisely identify precisely those patterns within a workload that firmware should be tuned to handle.

Our next step is to further improve the Distiller's library, and carefully examine the consequences of our design decisions. We first plan to add the attributes necessary to enable the Distiller to handle the block-level workloads generated by file servers. Next, we will examine the trade-offs between the precision used to measure attribute (e.g., the number of histogram bins for an empirical distribution or number of states for a Markov model) and the representativeness of the resulting synthetic workloads. Finally, we plan to examine the consequences of our design decisions. For example, we will evaluate how different search algorithms (e.g., choosing attributes using a first-fit vs. best-fit metric) affect the set of attributes chosen, the representativeness of the resulting workloads, and the running time of the Distiller.

After we have expanded the Distiller as described above, we plan to examine the key attributes (as determined by

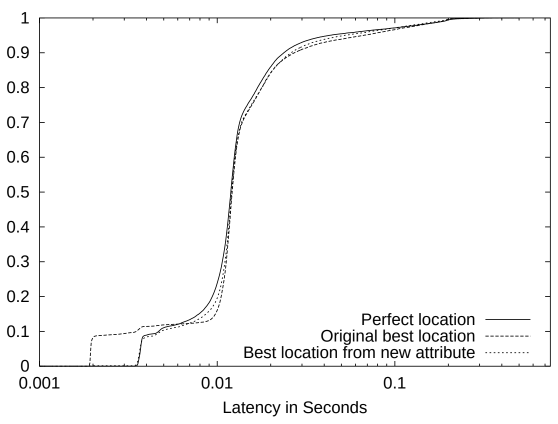

Figure 10. The new "Jump distance within state" attribute captures the location-related behavior of the DSS workload better than the existing $\{$location$\}$ attributes.

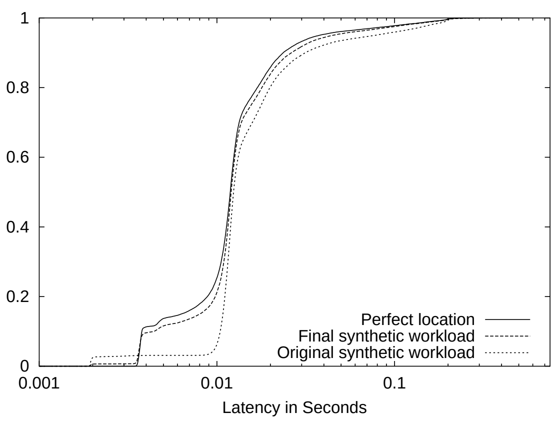

Figure 11. This graph shows the performance of the initial and final synthetic workloads based on the information distilled from the DSS trace.

the Distiller) of many different workload, storage system combinations. First, we will examine how the set of attributes chosen by the Distiller changes with respect to small changes the storage system (e.g., the prefetch length, cache size, or LU configuration). This investigation is important, because any synthetic workload used to compare several design decisions must be representative with respect to all designs/configurations under consideration.

Then, we will compare the set of attributes chosen by the Distiller for several traces of the same system taken several years apart and determine how a workload's set of key attributes changes over time. Privacy concerns discourage system administrators from collecting traces and making them available to the public. Some administrators are willing to make older traces available to the public. However, these traces are often not representative of current workloads, and are, therefore, of limited use as input to evaluations of modern storage systems. If the set of key attributes remains constant over time, and the corresponding attribute-values change predictably, we may be able to use traces of older workloads to generate synthetic workloads that represent current workloads.

Long term future work includes improving the Distiller's algorithm for selecting a specific attribute once an attribute group is chosen, proving the optimality of the attributes chosen by the Distiller (i.e., that the set of attributes is as small as possible, or that the set of attribute-values is as compact as possible given the desired degree of accuracy), and improving the quality of the hints the Distiller gives when the library is insufficient.

Finally, we believe that the Distiller will work given any reasonable quantification of storage system behavior (e.g., utilization or power consumption [26]). Furthermore, we believe that these techniques are generalizable to other areas of computer systems design (e.g., generation of synthetic processor and/or memory access traces). We would like to generate synthetic workloads with respect do different storage system behaviors, and apply the Distiller to these other domains.

8. Conclusions

This paper describes the design and evaluation of the *Distiller*, a tool that automatically extracts a set of attributes that specifies a representative synthetic I/O workload. The tool requires no human intervention, and is therefore inexpensive to run.

We built the Distiller around several key design principles: (1) It builds the set of key attributes iteratively, adding one attribute per iteration. (2) It takes a divide-and-conquer approach, including estimating the potential benefits of attribute groups, allowing it to evaluate the most important attributes first. (3) If its library of attributes proves insufficient, the Distiller can identify what relationships must be captured, thus helping to guide the invention of a new attribute. (4) Its extensible structure facilitates the incorporation of new attributes as they become available.

We demonstrated the execution of the Distiller on both artificial and production workloads. These examples highlight the key ideas on which the Distiller is designed and demonstrate that the Distiller can be an effective aid in the development of representative synthetic I/O workloads.

A. Synthetic workload generation

The Distiller's operation is independent of the tool used to generate synthetic workloads. It requires only that the generation tool be able to produce a synthetic workload with a specified set of attribute-values. However, because synthetic workload generation is often a non-trivial problem, we discuss a few of our generation techniques in this section.

Our synthetic workload generation tool, *GenerateSRT*, coordinates the operation of several individual generators. Each generator is responsible for reproducing a single attribute. Two generators that produce values for the same parameter will interfere with each other; therefore, a workload may be specified using at most one attribute from each attribute group.

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

Some generators corresponding to attributes in two-parameter attribute groups produce values for only one of the corresponding request parameters. For example, the generator that produces separate distributions of location for read requests and write requests generates only location values. We can use any {operation type} generator the operation types, then use this {operation type, location} generator to produce locations based on the current operation type. Notice, however, that not every {operation type, location} attribute is compatible with (i.e., will not interfere with) every {location} attribute. For example, most {location} attributes will be incompatible with a joint distribution of operation type and location. Each attribute includes a list of the corresponding generator's dependencies and restrictions. The Distiller is able to use this information to selects sets of attributes that will not interfere.

The remainder of this section discusses several of our generation techniques.

A.1 Run counts

To generate the desired distribution of run lengths, we simply choose the head and length of each run from the given distributions. The distribution of locations that form the head of runs may be different from the distribution of all locations; therefore, the run length analyzer should be designed accordingly. When using this method, it is possible to specify a run that contains invalid locations. This happens so infrequently that we simply truncate these runs.

A.2 Jump distance

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

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

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

We suspect (but have not yet proved) that an exact solution to this problem is NP-Complete.[6] Consequently, we generate an approximate solution using the greedy algorithm in figure 12:

This algorithm will maintain the desired distribution of location values (because only location values on the list are chosen); however, it is not guaranteed to maintain the distribution of jump distance. If no valid jump distances are found for a particular location, the algorithm randomly chooses a location from the given distribution (it does not back-track); therefore, in pathological cases, the jump distance distribution could differ significantly from the desired distribution. Fortunately, we have found that, in practice, the resulting distribution is close enough to produce a representative synthetic workload.

A.3 Markov model

The generator corresponding to the conditional distribution and transition matrix attributes discussed in section 3.3 is very straightforward. It chooses the current state, then draws a specific value from the appropriate conditional distribution. If a transition matrix is specified, the current state is determined by following the randomly chosen transition. If no transition matrix is specified, then the current state is calculated from the values of previously chosen parameters. For example, MM(operation type, location, 100, 1) does not require a transition matrix when the {operation type} generator is applied first.

[6]Furthermore, because the set of locations and jump distances are drawn randomly from a distribution, it is possible that there is no exact solution.

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

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

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

while (! list_empty(location))
{

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

    set_location(proposed_location)
    remove_from_list(jump_distance, jd_index);
    remove_from_table(proposed_location);
}
```

Figure 12. Greedy algorithm for choosing jump distances

When using several Markov model generation techniques simultaneously, one must resolve any dependences and avoid "loops". For example, generating a workload based on the MM(operation type, location, 1, 1) attribute requires some means of generating operation type (e.g., MM(operation type, operation type, 1,5) or a simple read percentage). Circular dependencies such as MM(operation type location, 1,1) and MM(location, operation type, 1,1) will cause the workload generator to fail. The Distiller automatically resolves all dependencies and dependencies.

Similar techniques can be used to generate distributions of jump distance, run length, and head-of-run locations that depend on other parameters.

A.4 Run count within state

To generate a workload with the specified "run count within state" attribute, we separately generate the sequence of locations for each state. We then interleave these separate streams of locations according to a Markov transition matrix of location.

A.5 Jump distance within state

To generate a workload with the specified "jump distance within state" attribute, we separately generate the sequence of locations for each state. We then interleave these separate streams of locations according to a Markov transition matrix of location.

References

[1] 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.

[2] 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.

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

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

[5] T. M. Conte and W. W. Hwu. Benchmark characterization for experimental system evaluation. In *Proceedings of the 1990 Hawaii International Conference on System Sciences*, volume I, pages 6–18, 1990.

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

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

[8] 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.

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

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

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

[12] 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.

[13] 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.

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

[15] 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.

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

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

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

[19] 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.

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

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

[22] C. A. Thekkath, J. Wilkes, and E. D. Lazowska. Techniques for file system simulation. *Software—Practice and Experience*, 24(11):981–999, November 1994.

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

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

[25] 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.

[26] J. Zedlewski, S. Sobti, N. Garg, F. Zheng, A. Krishnamurthy, and R. Wang. Modeling hard-disk power consumption. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, pages 217–230. USENIX, March 2003.