Grand Valley State University
ScholarWorks@GVSU

Technical Library | School of Computing and Information Systems

2006

Generating a Jump Distance Based Synthetic Disk Access Pattern

Dr. Zachary Kurmas
Grand Valley State University, kurmasz@gvsu.edu

Follow this and additional works at: https://scholarworks.gvsu.edu/cistechlib

ScholarWorks Citation
Kurmas, Dr. Zachary, "Generating a Jump Distance Based Synthetic Disk Access Pattern" (2006). *Technical Library*. 133.
https://scholarworks.gvsu.edu/cistechlib/133

This Technical Report is brought to you for free and open access by the School of Computing and Information Systems at ScholarWorks@GVSU. It has been accepted for inclusion in Technical Library by an authorized administrator of ScholarWorks@GVSU. For more information, please contact scholarworks@gvsu.edu.

Generating a Jump Distance Based Synthetic Disk Access Pattern
Technical Report GVSU-CIS-2006-01
10 May 2006

Zachary Kurmas
Dept. of Computer Science
Grand Valley State University
kurmasz@gvsu.edu

Jeremy Zito, Lucas Trevino, and Ryan Lush
Dept. of Computer Science
Grand Valley State University
{zitoj,trevinol,lushr}@student.gvsu.edu

Abstract

As part of our on-going research into improving the quality of synthetic, block-level I/O workloads, we have developed an algorithm that generates a synthetic (i.e., random) disk access pattern based on both (1) a given distribution of sectors accessed and (2) a given distribution of "jump distances" between successive disk accesses. Generating a synthetic disk access pattern that maintains both distributions exactly is an NP-complete problem (similar to the Traveling Salesman problem). In this paper, we (1) discuss our approximation algorithm, (2) show that it runs in a reasonable amount of time, (3) show that it reproduces both distributions with reasonable accuracy, and (4) demonstrate its overall effect on the quality of block-level synthetic workloads.

1. Introduction

Storage systems, such as disk arrays, must be evaluated with respect to the workloads that will be issued to them. These evaluation workloads can come from one of two sources: (1) actual traces of existing storage systems in action, or (2) synthetic (i.e., random) workloads generated based on a set of desired workload properties (such as percentage of requests that are reads, mean request size, etc.). Both approaches have strengths and weaknesses: Traces are more accurate, but are large, inflexible, and difficult to obtain. Synthetic workloads are flexible and can be represented compactly, but are often inaccurate.

Because evaluations conducted using synthetic workloads often do not yield the same results as evaluations conducted using traces of "real" workloads, it can be difficult to apply results obtained using synthetic workloads to "real world" situations. Our long-term research goal is to improve the accuracy of synthetic workloads enough that storage system evaluators can use them in place of workload traces.

Improving synthetic workloads requires two steps: We must first determine which properties a synthetic workload should share with the "real" workload for which it will substitute (henceforth called the *target* workload). We must then develop algorithms to produce synthetic workloads that maintain the desired properties. We addressed step 1 previously by developing a tool, called the *Distiller* that automatically identifies those workload properties on which a synthetic workload should be based [12, 13]. The Distiller relies on a library of workload properties from which to choose. This paper addresses step 2 by adding a needed property called "Jump Distance" to the Distiller's library. The *jump distance* between two I/O requests is the difference between their starting sectors. This measurement approximates how far a disk's read/write head must travel before serving the next request.

Section 2 provides background and further motivates the use of synthetic workloads and our study of jump distance. Section 3 explains our approximation algorithm. Section 4 presents our results. Section 5 discusses future work; and Section 6 concludes.

2. Motivation and background

In this section, we discuss (1) the motivations for using synthetic workloads, (2) our previous research that led us to focus on jump distance, (3) reasons for basing synthetic workloads on jump distance, (4) the reasons generating a disk access pattern that includes jump distance is not trivial, and (5) related work.

2.1 Why synthetic workloads

Storage systems (and proposed storage system designs) can be evaluated only in the context of a specific workload [3,

4]. For example, most storage systems contain a cache that reduces the number of I/O requests that access the physical disks. A hypothetical, newly developed cache replacement algorithm may improve the performance of a storage system when it serves a database, but degrade the performance of the same system when used with a file server.

Many researchers use workload traces to evaluate storage systems. They monitor a storage system in a production environment, collect all the requests made of that system, then issue those requests to the system under test. This use of traces is beneficial because it is accurate: It indicates precisely how the system under test will perform when issued the traced workload.

Unfortunately, the use of traces has several limitations: (1) Workload traces are very large (from hundreds of megabytes to tens of gigabytes), which can make them difficult to obtain, store, and share. (2) Many system administrators are concerned that, even with the actual data itself removed, traces will reveal valuable information to competitors; therefore, they hesitate to make them publicly available. Finally, (3) traces are inflexible: It is difficult to modify them to represent expected future workloads.

The advantages of using synthetic workloads are that (1) they are compact. They can be specified using only the parameters to the algorithm that generates them (e.g., the desired percentage of reads). In contrast, a workload trace contains the individual values for each I/O request. A one-hour trace can contain millions of requests. (2) Because they are specified using only high-level parameters, synthetic workloads do not contain any specific information. We are hopeful that system administrators will be more willing to provide the parameters used to generate synthetic workloads than they are to provide entire, detailed workload traces. Finally, (3) synthetic workloads are flexible. For example, one can produce a synthetic workload with a higher percentage of read requests by simply changing that parameter. In contrast, simple methods of changing a workload trace's read percentage, such as changing write requests to read requests, may have unknown secondary affects on the evaluation results (e.g., if the set of sectors read to, and written from, are disjoint).

The challenge in using synthetic workloads is that they are rarely accurate. When used as part of a storage system evaluation, a perfectly accurate synthetic workload will produce the same results as the *target workload*. In order for a synthetic workload to be accurate, it must share certain properties with the target workload. In particular, it must share those properties that affect how the storage system will behave when serving the workload. For example, a synthetic workload must maintain not only the same I/O rate as the workload it models, but it must also maintain a similar number and intensity of bursts so similar queues of I/Os build up on the storage system under both loads.

2.2 The Distiller

Many existing workload synthesis techniques were developed for specific studies and are not generally useful. In response, we invented the Distiller to easily and automatically evaluate which existing generation techniques produce accurate synthetic workloads for new storage systems.

The Distiller does not "invent" workload properties. Instead, it takes as input a library of properties other researchers have already invented. The Distiller may conclude that no subset of its library will produce a sufficiently accurate synthetic workload. We must then add new properties to the library. The Distiller provides useful hints, but we must do most of the work. Based on its hints, we decided to add an improved jump distance property.

2.3 Jump Distance

Both our understanding of storage systems and the results of evaluating the Distiller led us to implement a jump distance-based disk access pattern generator. Logically, we expect the amount of disk head movement to affect disk array behavior because physical movement of any kind tends to produce bottlenecks in computer systems. Empirically, the Distiller found that our set of workload properties did not define a disk access pattern that was similar enough to the target workload to be useful. Simplistic means of measuring and reproducing disk head movement noticeably improved the quality of the resulting synthetic workloads. We believe further improvements in our techniques of measuring and reproducing disk head movements will lead to continued improvement in the quality of our synthetic workloads.

We use the difference between the starting sectors of successive I/O requests (i.e., $request[x].starting_sector - request[x-1].starting_sector$) as an approximate measure of disk head movement. We call this difference the *jump distance* between requests $x-1$ and x. At a high level, the jump distance is a measure of how far the disk head must physically move to reach the starting sector of the next request. When specifying a synthetic workload, we use the distribution of jump distance as one of the parameters.

Notice, however, that the distribution of jump distance is only an approximate measure of disk head movement:

- The actual head movement, if any, is from the *end* of request $x-1$ to the beginning of request x. (Most people use the term "jump distance" to refer to the difference between the *end* of request $x-1$ and the beginning of request x.)

- Requests $x-1$ and x may lie on the same track. This situation causes rotational delay instead of disk head movement.

- Request x may already be in cache, resulting in no disk head movement.
- In a large storage system, requests x and $x-1$ may be on separate physical disks.

These approximations either (1) have a small effect on the quality of the resulting synthetic workload, or (2) are compensated for by other aspects of the Distiller. During our evaluations of the Distiller, we found that the correlation between an I/O request's starting sector and its size had very little effect on disk array behavior, which suggests that calculating jump distance using starting sectors only is a reasonable approximation. Second, the precise effects of the jump (disk head movement or rotational latency) are not important, as long as both the synthetic and target workloads induce the same behavior on the storage system. Third, the Distiller is responsible for finding all necessary workload properties, including a property that will generate workloads with similar cache behavior. Thus, our jump distance generation algorithm need not explicitly address whether a request is in the cache. Finally, when generating a synthetic workload for a storage system with many physical disks, we can divide the address space according to the physical disks onto which each sector corresponds, then analyze the different sections of the address space separately.

2.4 Challenges

In addition to the distribution of jump distances, an accurate synthetic workload must also maintain the distribution of sectors used by the workload being modeled. Today, typical storage systems contain over a billion addressable sectors (i.e., a terabyte or more of data); however, many workload traces access only a small fraction of those sectors during the course of an evaluation. If we were to generate a synthetic disk access pattern without regard to the actual distribution of sectors accessed, the synthetic workload would likely have much different cache behavior than the target workload it is intended to model.

The problem of generating a list of sectors accessed that maintains both a given distribution of sectors accessed and a given distribution of jump distances between successive disk accesses reduces to the NP-complete Hamiltonian Path problem (a form of the Traveling Salesman problem). As a result, we must find a sufficiently accurate approximation algorithm.

These challenges illustrate another use for the Distiller: As the number of desired workload properties increases, so do the difficulties associated with generating a synthetic workload that maintains all desired properties (which is why synthetic workloads do not simply maintain every imaginable property). The Distiller helps alleviate this problem by (1) limiting the number of properties that must be maintained simultaneously, and (2) attempting to choose properties that can easily be maintained simultaneously.

2.5 Related work

The literature describes and evaluates many techniques for generating synthetic block-level I/O workloads [5, 6, 7, 8, 9, 16, 17], file-level workloads [1, 2, 15] and application-level I/O workloads [10]. Most focus on accurately synthesizing the workload's arrival pattern. Two of the few that focus primarily on block-level disk access patterns are Gomez's On/Off technique and Wang's PQRS model.

Gomez et al. generate a synthetic disk access pattern by mimicking the behavior of individual processes [6, 7]. Each process is assumed to be very active for an "on" period, then inactive during an "off" period. During its "on" period, a process chooses the starting sector of each I/O in one of three ways (1) sequential to the previous I/O, (2) equal to the starting sector for the first I/O in the current "on" period, or (3) spatially local to (i.e., within 500 sectors of) the starting sector for the current "on" period. This generation technique assumes that the trace of the I/O workload being modeled identifies the process that issued each request and that these processes generate I/Os in an on/off pattern. In contrast, our technique requires only the list of requests made to the storage system. It also does not make any assumptions about the behavior of the workload it is attempting to reproduce.

Wang, et al., developed an attribute that "captures all the characteristics of real spatio-temporal traffic" [16]. In other words, this technique strives to not only reproduce the burstiness of the access pattern and the arrival pattern, but also to reproduce the correlations between them. The PQRS algorithm measures four parameters (p, q, r, and s) that are based on the joint entropy of the sectors accessed and arrival time values. The corresponding generation technique then uses these values to recursively construct a joint distribution for sectors accessed and arrival time. To fit with the Distiller's model of workload analyzers and generators, we desire at present a technique that maintains disk access patterns only.

Hong, Madhyastha, and Zhang developed a generation technique that chooses several intervals of a real trace to represent the entire trace. The algorithm then generates a complete synthetic trace by concatenating copies of those intervals [8, 9].

This technique is highly effective in generating a synthetic arrival pattern; however it fared poorly when generating the disk access pattern also, because repeating the intervals chosen to represent the entire trace tended to add too much temporal locality to the workload. We expect that combining our jump distance-based generator will help address the limitations of Hong's clustering-based technique.

3. Our algorithm

Our problem is to take as input (1) the distribution of sectors accessed by the target workload, (2) the desired distribution of jump distances within the target workload, and (3) the desired number of I/Os; then produce as output a synthetic disk access pattern.

Our approach is to transform the generation problem into an instance of the Hamiltonian Path problem, then apply a brute-force, depth-first search. The resulting path defines the sequence of starting sectors. If the search does not find a Hamiltonian path within a reasonable amount of time, we take the longest path found and use approximation techniques to complete the access pattern. The approximation techniques do not produce an exact solution (i.e., a sequence of sectors that precisely maintains the desired distributions); however, Section 4 will show the approximate solution is usefully accurate.

We begin by generating a list of I/O requests whose starting sectors exhibit the distribution given as input. This distribution is presented as a histogram — an array in which the value at index x (i.e, $h[x]$) tells what fraction of I/O requests have x for a starting sector. To generate $num_requests$ I/O requests with the same distribution, we create $h[x] \cdot num_requests$ I/O requests for each value of x. The problem is now to find an ordering of the requests that exhibits the distribution of jump distances specified by the input. This problem is similar to the famous Hamiltonian Path problem.

The Hamiltonian Path problem is as follows: Given a graph G, do there exist two vertices v_a and v_b such that there is a path from v_a to v_b that visits each vertex in G exactly once? This problem is NP-complete; the fastest known algorithm is not much faster than a depth-first-search through the set of possible paths.

A depth-first search for a Hamiltonian path would proceed as follows: First, order the vertices. (We'll label them v_1 through v_n.) Second, choose a starting vertex. (We'll choose v_1.) Now, build the path iteratively. During each iteration, choose the adjacent vertex with the lowest index that does not yet appear in the path. For example, if v_1 is adjacent to vertices v_3, v_6, and v_{10}, choose v_3 and append it to the path. If at any point, all the vertices adjacent to the vertex currently at the end of the path already appear in the path, backtrack. For example, if there were no previously unvisited vertices adjacent to v_3, we would remove v_3 from the end of path and replace it with v_6 (because of all vertices adjacent to v_1, v_6 has the next-lowest index).

We cast our generation problem as a Hamiltonian Path problem on a directed graph. We add one vertex for each I/O request. We then add an edge between two vertices if the jump distance between them appears in the jump distance histogram. For example, given accesses d_1 with starting sector 47 and d_2 with starting sector 59, we add the edge (d_1, d_2) if, and only if, the jump distance histogram bin 12 is not empty.

We can now search for an ordering of the requests by choosing an initial I/O request (call it d_1), then executing a depth-first-search of possible paths beginning with d_1. If we find a valid path of length n, we have found an ordering of I/O requests that maintains the desired jump distance distribution. Unlike the Hamiltonian Path problem, however, we must not only check that we use each I/O request (i.e., vertex) only once;[1] but, we must also be sure we do not use a given jump distance more often than specified by the histogram. In other words, even though there may be many edges in the graph that correspond to a jump distance of j, we must make sure that the number of such edges used does not exceed the fraction specified by the input. (In this regard, our problem is not strictly a Hamiltonian Path problem — it is a "harder" problem.[2])

When performing a depth-first search, we must specify the order in which our algorithm examines the I/O requests. We consider five orderings:

1. **Ascending:** Examine the I/O requests in order from smallest to largest starting sector.
2. **Ascending absolute value:** Examine the I/O requests closest to the current sector first.
3. **Descending:** Examine the I/O requests in order from largest to smallest starting sector.
4. **Descending absolute value:** Examine the I/O requests farthest away first.
5. **Random:** Assign a random order.

Two factors make our brute-force approach reasonable given the theoretical complexity of the problem: First, not every Hamiltonian Path problem requires exponential time. For example, it is trivial to find a Hamiltonian path in a complete graph (a graph in which every pair of vertices are adjacent), regardless of its size. We are hopeful that our jump distance problems will define graphs whose structures lend themselves to quick solutions. Second, a solution need not be exact to be useful. If the workload we are attempting to model has 10% of its accesses with a jump distance of 1 sector, it is acceptable for our solution to have slightly more or fewer. We will see that this flexibility allows us to find a solution considerably faster (in practice, not in theory).

We employ two main techniques for obtaining approximate solutions quickly:

[1] Notice that we can use each I/O request once only; however, each sector may be accessed by more than one I/O request. Thus, sectors may be used more than once.

[2] Both versions of the problem are NP-complete; thus, our version is not "harder" in the formal sense.

1. We complete the path randomly after a given amount of backtracking, and
2. We allow a limited number of sectors and/or jump distances to be used more often that specified by their respective histograms.

In the first case, we specify how many times the depth-first search may backtrack without finding a longer valid path. (We call this parameter "*max stall*".) If the search backtracks too often, we terminate it, take the partially completed path, and place the remaining unused I/O requests randomly throughout the path. This maintains the desired distribution of sectors visited, but can produce a jump distance histogram that differs from the one specified.

In the second case, we specify "approximation factors" for sectors used and jump distance (for example, 10% and 15% respectively), and allow a sector to be used 10% more often than specified by the distribution of sectors accessed. Similarly, we allow a given jump distance to be used 15% more often than specified by the jump distance distribution. This technique allows the search to find longer paths, but produces a solution where both the distribution of sectors used and the distribution of jump distances differ from the distributions given as input.

In the next section, we will evaluate how our choices for max stall and the approximation factor affect the quality of our synthetic disk access patterns.

4. Experimental results

We evaluated our synthetic disk access pattern generator using two different target workloads: E-mail and OLTP. Both workloads are divided into *logical units* (LUs) — sets of sectors that appear to clients as a separate storage systems. The E-mail workload contains 22 LUs; OLTP contains 38. For this paper, we analyze and generate each LU separately. Analyzing each LU separately makes sense when studying jump distance because only those jumps between requests to the same physical disk have a significant affect on storage system behavior. In addition, analyzing the LUs separately helps reduce the size of the resulting graphs.

E-mail: Our first workload to model is a one-hour trace of the workload created by the OpenMail e-mail application. It has a mean request rate of 304 I/Os per second and a mean throughput of approximately 2.33MB/s. The complete workload is described in detail in [11]. This workload's distribution of sectors accessed makes it an interesting example with which to test our generator. Over 78% of the I/Os are write requests. Among the write requests, almost half access a small percentage of the sectors. The remaining writes and all the reads are distributed much more evenly over the sectors accessed. We suspect this distribution reflects differences in the accesses to the portion of the disk array used to store meta-data, and the portion used to store individual messages.

OLTP: Our second workload to model is a 1994 on-line transaction processing (OLTP) trace that measures HP's Client/Server database application running a TPC-C-like workload at about 1150 transactions per minute on a 100-warehouse database. It has a mean request rate of 537 I/Os per second and a throughput of approximately 1.35 MB/s. Approximately 50% of the requests are reads. This workload is interesting to study because several of the LUs contain the database's log. Accesses to these LUs tend to be highly sequential. In addition, this workload's higher read percentage and I/O rate provide a useful contrast to the E-mail workload.

In this section, we present four analyses. First, we evaluate how many I/O requests we can analyze and generate given the 512MB of RAM on the machines available to us. Next we evaluate how run time, input size, and vertex ordering affect the length of the partial Hamiltonian path found. Third, we evaluate the quality of the synthetic disk access patterns with respect to the input distributions. Finally, we evaluate the quality of the synthetic disk access patterns with respect to the accuracy of the resulting synthetic workload.

For the experiments below, we have our analyzer (the tool that produces the input histogram) analyze as many I/Os as we request be generated. For example, when generating 25,000 requests, we analyze the first 25,000 requests of the workload being modeled. The characteristics of many production workloads change slightly from minute to minute. As a result, we do not expect a synthetic workload of length x to have behavior identical to the target workload of length y. Therefore, it is difficult to evaluate the accuracy of the synthetic workload unless it models a production workload of the same length.

4.1 Memory requirements

Our algorithm defines a graph that grows as much as quadratically with respect to the number of I/Os analyzed. Before designing other experiments, we must first investigate how many I/Os we can analyze in a timely manner using our equipment.

Figure 1 shows that each of the E-mail workload's 22 LUs require at most 428MB of RAM to store the graph. Other data structures bring the entire memory requirements to just under 800MB — a reasonable size given today's technology. In contrast, Figure 2 shows that at least one of the OLTP workload's LUs needs more than 2GB of RAM to analyze and generate only 32768 requests. Only 6 of 38 LUs require less than 512MB of RAM to analyze 32768 I/Os, and only 6 require fewer than 2GB to analyze 65536 I/Os. In fact, analyzing the entire OLTP workload requires

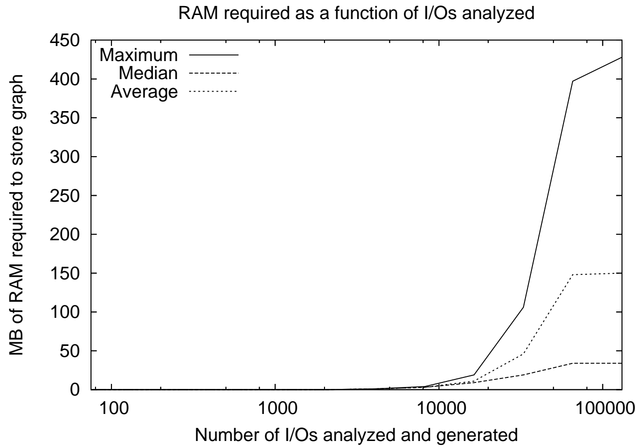

Figure 1. No E-mail LU requires more than 428MB of RAM to store the edges in the graph.

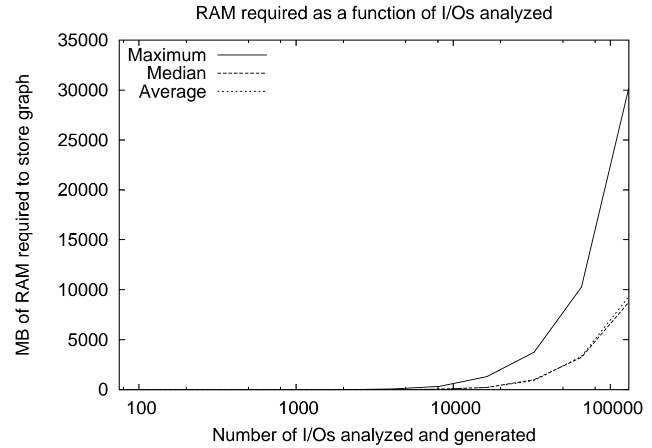

Figure 2. With only 4GB of RAM, we cannot analyze more than 32,000 OLTP I/Os at one time.

30GB of RAM just to store the graph!

To address the large memory requirement of the OLTP workload, we either (1) examine at most the first 16384 I/Os from each LU, or (2) divide each LU into 6 equal parts. No part of any LU requies more than 2GB of RAM. We divide the workload by sector number: I/Os with sectors in the 16th percentile, I/Os with sectors in the 16th through 33rd percentile, I/Os with sectors in the 34th through 40th percentile, etc. This division makes more sense than dividing the workload by time because jumps between I/Os in different parts are large enough to require substantial disk head movement that need not be maintained precisely. All jumps between two different parts affect the disk approximately the same way, whereas jumps between I/Os in the same part can have very different effects depending on whether the jump keeps the disk heads on the same track and/or cylinder.

4.2 Length of partial path

Because the Hamiltonian Path problem is NP-complete, it is unlikely our depth-first search will find a complete Hamiltonian path in a reasonable amount of time. Three factors affect how long of a partial path is generated: (1) the length of the trace analyzed, (2) the amount of time we allow the depth-first search to run, and (3) the order in which the algorithm examines vertices.

4.2.1 Input size

We had originally hoped that the graph defined by the input would be dense enough to contain many Hamiltonian paths, one of which would be quickly discovered by a brute-force search. Figures 3 and 4 show that this is not the case. We ran our generation algorithm several times requesting increasingly long sequences of I/Os. We allowed the algorithm to run until it had not made any progress during the previous 10 million iterations. At that point, we terminated the algorithm and measured the length of the longest path created and reported that length relative to the desired number of I/O requests. We ran the algorithm separately on each LU. Figures 3 and 4 show the maximum, minimum, mean, and median length of the longest path over all LUs and vertex orderings. (When analyzing the use of the random vertex ordering, we took the average over 10 executions using 10 different random seeds.) We can see that, for any LU or vertex ordering, the algorithm is not able to complete quickly once the desired number of I/Os reaches about 150.

Notice that the length of the partial path found (relative to the number of I/Os desired) dips for lengths of approximately 1,000 I/Os, then rises again. Figures 5 and 6 show that the density of the graphs we are searching is correlated to the length of the path found. This correlation makes sense because it is easiest to find Hamiltonian paths in extremely dense and extremely sparse graphs.

4.2.2 Vertex ordering

The order in which the depth-first-search examines vertices in the graph affects how long the algorithm takes to find a Hamiltonian path. If the vertices are searched in an extremely fortuitous order, then the algorithm will find a Hamiltonian path without any backtracking. We experimented with five different orderings (explained in Section 3). Figures 7 and 8 show how the vertex ordering affects the length of the path that is generated before the algorithm goes 10 million iterations without producing a longer path.

With the OpenMail workload, the random vertex ordering does best when the desired number of I/Os is large –

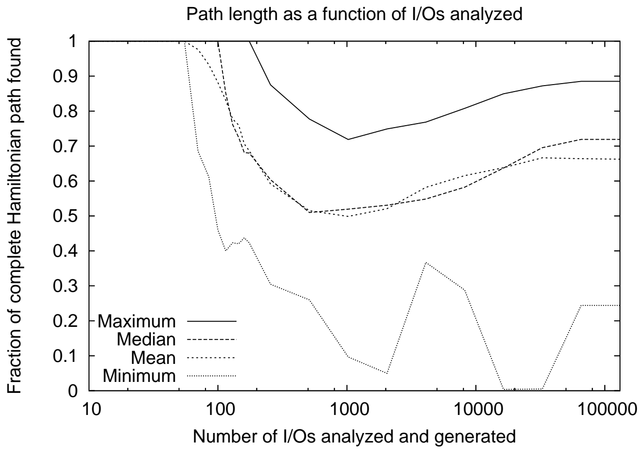

Figure 3. Our brute-force approach completes only when we desire fewer than 150 E-mail I/Os.

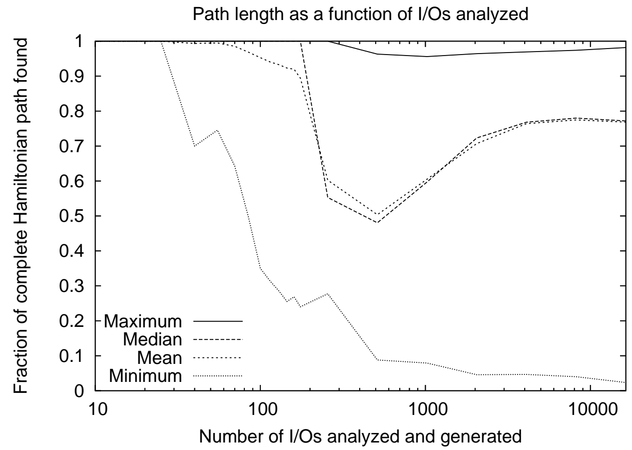

Figure 4. Our brute-force approach completes only when we desire fewer than 150 OLTP I/Os.

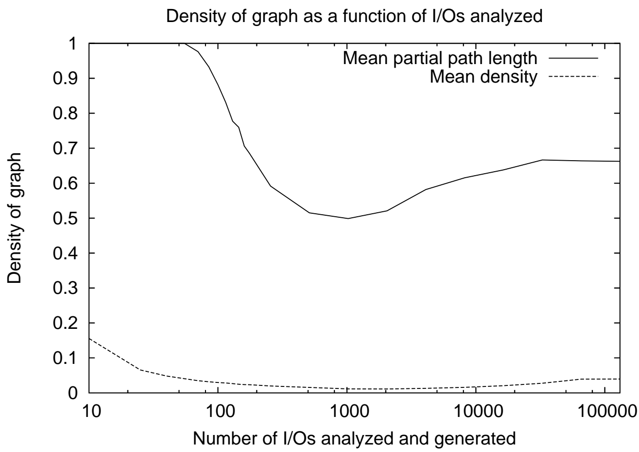

Figure 5. Higher density produces longer partial Hamiltonian paths (E-mail).

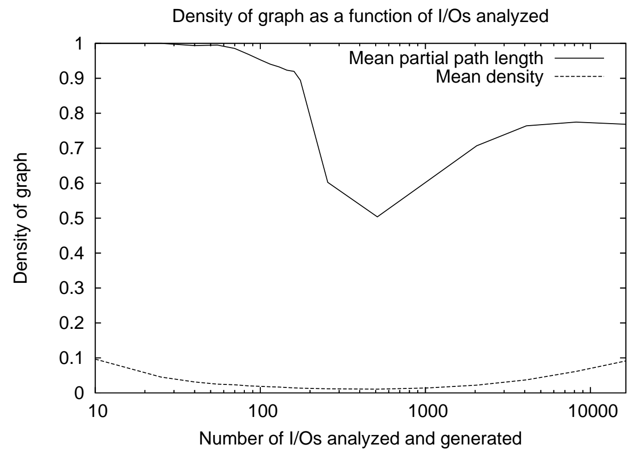

Figure 6. Higher density produces longer partial Hamiltonian paths (OLTP)‘.

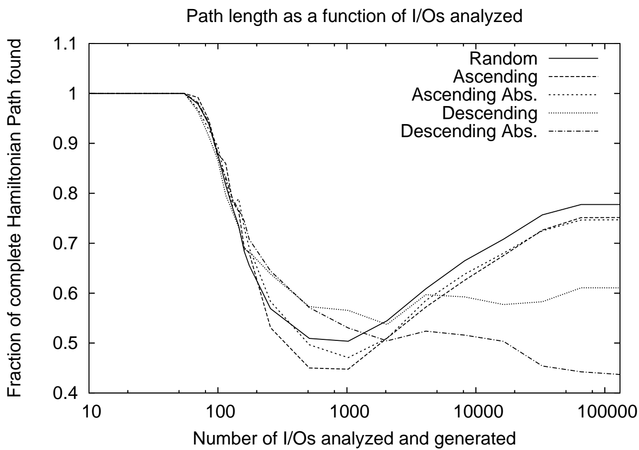

Figure 7. A random vertex order produces the longest partial Hamiltonian path for E-mail.

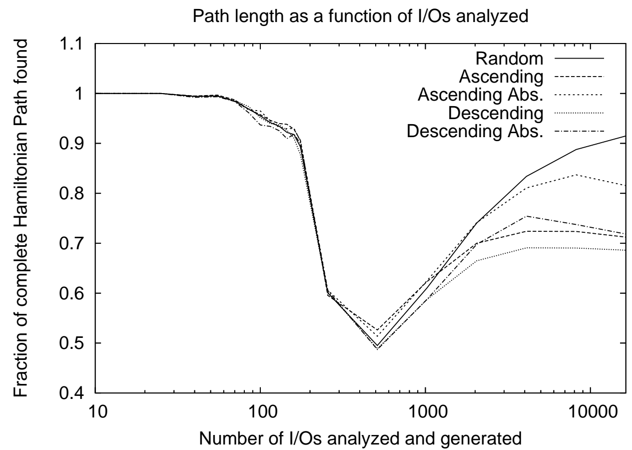

Figure 8. A random vertex order produces the longest partial Hamiltonian path for OLTP.

slightly better than both ascending sorts. We suspect the random vertex order leads to the longest partial Hamiltonian paths because it is less likely to "use up" sectors and jump distances early. ("Using up" sectors and jump distances causes backtracking.) For example, when configured to use the "Ascending absolute value" sort, the depth-first search always considers the I/O requests with the current starting sector first. As a result, the depth first search begins building a path by repeating the initial sector until either (1) that sector is used as many times as specified in the input distribution, or (2) the jump distance 0 is used as many times as specified in the input distribution. Soon afterward, the jump distance of 1 sector is "used up". Small jump distances are easy to match (i.e., there are more pairs of requests 1 sector apart than there are pairs of sectors 2,435,812 sectors apart); thus, the "ascending order" searches build long paths quickly. However, once the common jump distances have been used, the rate of progress slows. In contrast, by examining vertices in a random order, the depth-first search is more likely to consider less common jump distances first. The "descending order" sorts also have this property; but because larger jumps are harder to match, the search makes less progress before reaching the backtracking limit.

4.2.3 Maximum Stall

Because the depth-first search cannot find an exact answer in a reasonable amount of time, we must terminate the algorithm at some point and complete the list of I/Os using approximation techniques. For our first set of experiments, we stopped the algorithm after it had not made any progress in 10 million iterations. Upon further investigation, however, we found that we can achieve almost the same result while terminating the algorithm much earlier. When generating full E-mail LUs, on average, the path after 100,000 stalls was 99% the length of the path after 10 million stalls. Furthermore, among all LUs, orderings, and random seeds, only 1.5% of the experiments produced a path after 100,000 stalls that was less than 90% the length of the path after 10 million stalls. Similarly, when generating 8192 OLTP I/Os, on average, the path after 100,000 stalls was 99.5% the length of the path after 10 million stalls.

From these results, we have chosen to run our depth first search only until it goes 1 million iterations without producing a longer path. (We use 1 million instead of 100,000, because the difference in running time is nominal compared to the time needed to read in the data and generate the graph.) This limit will produce a reasonable lower-bound on our results.

4.3 Distribution quality

Our results from the previous section show that we cannot use the depth-first search alone to generate a synthetic disk access pattern. In this section, we evaluate two methods of creating a complete synthetic disk access pattern: "finish randomly" and "approximation factor."

In both cases, we evaluate the quality of the resulting synthetic disk access pattern by comparing its distribution of jump distance to the input from which it was generated. We quantify the difference between two distributions of jump distance using the weighted average of the percent difference between each pair of corresponding histogram bins. For example, we calculate the percent difference between bin x for the target workload and bin x for the synthetic workload as follows: $(\frac{|h_{target}[x]-h_{synthetic}[x]|}{h_{target}[x]})$. Because there is no universally accepted metric for quantifying the difference between distributions, we considered five different metrics. All five exhibited similar trends; so, in the interest of space and clarity, we choose to present only the average bin difference.

4.3.1 Finish randomly

As described in Section 3, the first option for generating a complete disk access pattern is to wait until the depth-first-search has not made any progress for the specified number of iterations, then generate the complete disk access pattern from the partial path by placing the unused I/O requests randomly throughout the path.

Figure 9 shows how the quality of our synthetic disk access patterns compare to a disk access pattern produced by choosing the sequences of sectors randomly without considering jump distance. Three of our five vertex orderings are considerably more accurate than the "completely random" access pattern. The descending and descending absolute value sort orders produce progressively lower quality access patterns as the length of the generated trace increases. We suspect this lower quality is a result of the correspondingly short partial paths from which the final disk access patterns are derived. (See Figures 3 and 4.)

4.3.2 Approximation factor

As described in Section 3, our second option is to allow the depth-first search to be a little "sloppier". In particular, we allow it to use sectors and/or jump distances more often than they appear in the target workload. We investigated different combinations of sector and jump distance approximation factors to see (1) which combinations allowed our depth-first search to finish within a reasonable amount of time and (2) how the synthetic disk access patterns produced compared in quality to those produced in Section 4.3.1.

Tables 1 and 2 show typical results for the E-mail workload. For these experiments, we generated 50,000 I/Os per LU using the random and ascending vertex ordering respec-

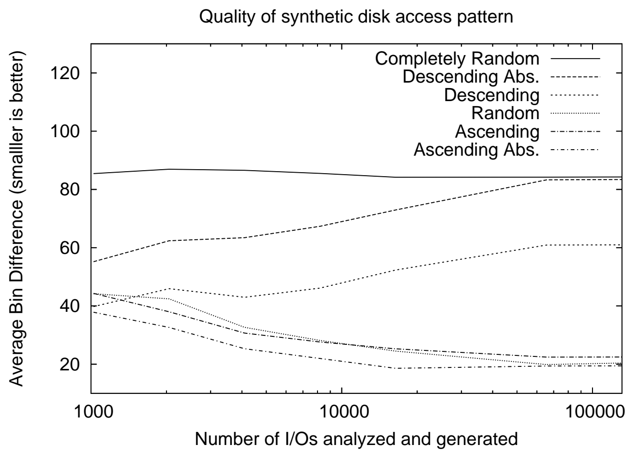

Figure 9. Ascending and random vertex orderings produce best quality synthetic disk access patterns for E-mail workload.

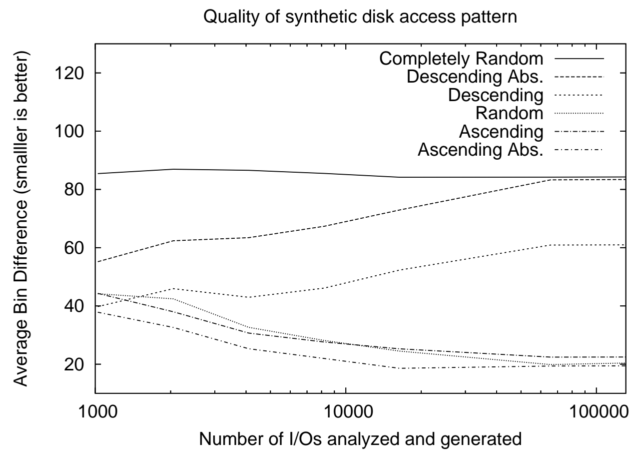

Figure 10. Ascending abs. and random vertex orderings produce best quality synthetic disk access patterns for OLTP workload.

Jump Dist.	Sector Count Factor							
Factor	1.00	1.01	1.05	1.10	1.25	1.50	1.75	2.00
1.00	77.4 (18.6)	77.4 (18.7)	77.6 (18.3)	77.9 (18.0)	81.2 (15.0)	86.5 (10.5)	87.8 (9.5)	91.9 (6.6)
1.01	77.1 (19.1)	77.2 (19.1)	77.8 (18.1)	78.4 (17.7)	81.3 (15.0)	86.6 (10.7)	88.0 (9.5)	92.1 (6.5)
1.05	77.8 (18.9)	77.8 (18.8)	78.1 (18.5)	78.7 (17.9)	81.8 (15.2)	87.2 (11.0)	88.8 (9.8)	92.5 (7.5)
1.10	77.7 (19.6)	77.7 (19.7)	78.4 (18.9)	79.1 (18.3)	82.6 (15.5)	88.1 (11.8)	89.8 (10.6)	93.8 (8.0)
1.25	79.7 (19.8)	79.7 (19.7)	80.5 (19.3)	80.1 (20.0)	85.4 (17.0)	91.1 (13.7)	92.8 (12.9)	95.9 (12.6)
1.50	83.5 (19.8)	83.4 (19.9)	84.0 (19.8)	84.5 (20.2)	88.9 (19.5)	95.1 (18.2)	98.0 (20.1)	99.4 (20.1)
1.75	84.2 (23.0)	84.3 (23.0)	85.5 (22.7)	87.0 (23.0)	91.8 (22.0)	98.3 (25.1)	98.9 (25.8)	99.6 (25.8)
2.00	91.0 (23.8)	91.1 (23.8)	91.6 (24.2)	92.3 (24.2)	98.5 (23.3)	100.0 (24.4)	100.0 (24.8)	100.0 (24.6)

Table 1. Length and quality of synthetic E-mail disk access pattern generated using approximation factors and a random vertex ordering. The first number is the length of the partial path found (as a percentage). The number in parentheses is the quality of the jump distance distribution.

Jump Dist.	Sector Count Factor							
Factor	1.00	1.01	1.05	1.10	1.25	1.50	1.75	2.00
1.00	74.7 (22.6)	74.8 (22.5)	75.1 (22.2)	75.7 (21.6)	78.5 (18.3)	83.1 (12.5)	84.7 (10.5)	88.2 (8.0)
1.01	74.9 (22.7)	74.9 (22.7)	75.1 (22.4)	75.9 (21.7)	78.6 (18.5)	83.2 (12.6)	84.9 (10.6)	88.2 (8.2)
1.05	75.4 (23.2)	75.5 (23.2)	75.9 (22.9)	76.5 (22.3)	79.1 (19.3)	84.0 (13.4)	85.8 (11.2)	89.0 (8.8)
1.10	76.3 (23.9)	76.3 (23.9)	76.8 (23.6)	77.5 (23.1)	80.1 (20.4)	84.9 (14.4)	86.9 (12.1)	90.3 (10.0)
1.25	79.1 (26.8)	79.2 (26.8)	79.7 (26.5)	80.5 (25.7)	83.8 (23.4)	88.8 (18.4)	90.2 (17.0)	93.1 (16.6)
1.50	83.9 (33.2)	84.1 (33.1)	84.9 (32.6)	86.4 (31.9)	90.9 (31.2)	95.0 (33.6)	96.4 (33.7)	99.1 (34.4)
1.75	86.1 (40.1)	86.2 (40.1)	87.0 (39.6)	88.6 (38.8)	93.3 (38.8)	98.3 (44.7)	99.0 (46.4)	100.0 (48.4)
2.00	92.7 (59.5)	92.8 (59.4)	93.9 (59.6)	95.0 (60.2)	98.3 (61.1)	99.8 (62.7)	100.0 (62.8)	100.0 (68.4)

Table 2. Length and quality of synthetic E-mail disk access pattern generated using approximation factors and an ascending vertex ordering. The first number is the length of the partial path found (as a percentage). The number in parentheses is the quality of the jump distance distribution.

tively. As expected, raising either approximation factor increases the length of the partial path found. However, the increases in length were less than we had expected. The depth-first search does not find a 99% complete Hamiltonian path until at least one of the approximation factors is 1.5. Fortunately, with the random ordering, increasing the approximation factor decreases the quality of the synthetic disk access pattern by at most 50%. In contrast, increasing the approximation factors when using the ascending ordering quickly decreases the quality of the resulting jump distance distribution until it is almost as low as if the sectors were ordered randomly without regard to jump distance.

4.4 Synthetic workload quality

Measuring the quality of the output distributions is useful because it tells us how well our generation algorithm reproduces the pattern it is designed to reproduce. However, the output distributions are merely "cosmetic" similarities between the target and synthetic workloads: They tell us nothing about the usefulness of the resulting synthetic workload. It is like expecting two people with the same height to have the same basketball skills.

In this section, we measure the usefulness of our synthetic disk access generator in producing accurate synthetic workloads. We will issue both the target and synthetic workloads to a storage system and measure the resulting distribution of response time (the percentage of I/Os that complete quickly and the percentage of I/Os that complete slowly).

We will use three different metrics to quantify the difference between the performances of two distributions:

1. **Mean response time:** The mean response time (MRT) is the simplest, but least accurate metric. A synthetic workload should have a mean response time similar to the workload it models; however, two workloads with very different behaviors can have similar mean response times. We express this metric as the percent difference between the mean response times of the compared workloads.

2. **Root-mean-square:** The root-mean-square (RMS) is the metric used in the related work (e.g., [5], [14]). Specifically, RMS is the root mean square of the horizontal distance between the response time cumulative distribution functions (CDFs) for the synthetic and target workloads. We normalize this metric by presenting it as a percentage of the mean response time of the target workload.

   The RMS metric is useful because it measures similarity based on the performance observed by the user. However, because the RMS metric sums the square of horizontal differences, workloads whose CDFs have horizontal "plateaus" tend to have very large RMS values — especially when those plateaus are near 1 on the y-axis (i.e., "heavy tails").

3. **Log area:** The log area metric is the area between two CDFs plotted with a log scale on the x-axis. Using the log scale causes differences at all percentiles to be weighted equally. This metric more accurately reflects the similarity of the workload's overall performance, but de-emphasizes differences most noticeable to the user.

Workload	MRT	RMS	Log area
Completely Random	15%	32%	6%
Ascending	1%	3%	1%
Ascending Abs.	9%	18%	4%
Descending	3%	6%	2%
Descending Abs.	13%	27%	6%
Random	11%	23%	5%

Table 4. Performance of synthetic E-mail workloads

Workload	MRT	RMS	Log area
Completely Random	4%	5%	9%
Ascending	25%	22%	69%
Ascending Abs.	20%	18%	43%
Descending	21%	20%	64%
Descending Abs.	9%	8%	16%
Random	3%	3%	4%

Table 5. Performance of synthetic OLTP workloads

We use the Pantheon disk array simulator to simulate the execution of our workloads [18]. Pantheon simulates disk arrays comprising several disks connected to one or more controllers by parallel SCSI busses. The controllers have large non-volatile-RAM caches. (This general architecture is similar to the FC-60 used in [12].) Pantheon provides many additional configuration parameters including number and type of component disks, and size of cache. Table 3 provides the Pantheon configuration used to study each workload.

In order to observe the effects of our synthetic disk access pattern only, we modified the target workloads. The modified E-mail workload is read-only, has a constant request size of 8KB, and issues precisely 328 I/Os every second. The modified OLTP workload is also read-only , has a constant request size of 2KB, and issues precisely 500 I/Os every second. Attempting to evaluate our disk access pattern in the context of the original workloads is difficult because the correlations between sectors accessed and whether those accesses are reads and writes have a large effect on storage systems with a write back cache (like the FC-60). Fixing the values for those I/O parameters not under study eliminates this problem [5].

Table 4 shows the performance of our modified E-mail

workload	Num disks	Num busses	Num raid groups	Disk size	disk type	Cache size	Bus rate
OpenMail	180	4	45	9 GB	Seagate Cheetah 10K rpm	1 GB	40 MB/s
OLTP	80	2	40	1 GB	Wolverine III (HPC2490A)	256 MB	40 MB/s

Table 3. Summary of Pantheon configurations

Workload	Hit rate
Target	67.0%
Completely Random	66.6%
Ascending	65.9%
Ascending Abs.	64.7%
Descending	65.4%
Descending Abs	65.0%
Random	64.3%

Table 6. Cache hit ratio for synthetic E-mail workloads

Workload	Hit rate
Target	8.7%
Completely Random	9.9%
Ascending	13.8%
Ascending Abs.	10.5%
Descending	10.5%
Descending Abs	8.9%
Random	7.3%

Table 7. Cache hit ratio for synthetic OLTP workloads

workload as compared to synthetic workloads generated using various sort orders. For comparison, we also show the performance of a workload in which the sequence of sectors accessed is chosen randomly without regard to jump distance.

We can see that, although using a random sort order produces the longest Hamiltonian path, it does not produce the best synthetic workload from a performance standpoint. We suspect this decrease in performance is caused by a loss of temporal locality. Upon examining Pantheon's log files, we found that the synthetic workload based upon a random sort order has a cache hit rate that is 2.7% lower than that of the target workload. (See Table 6.) In contrast, the most accurate synthetic workload — the one based on searching sectors in ascending order — has a cache hit rate closer to that of the target workload. Searching sectors in ascending order implicitly maintains temporal locality by considering the same sectors first. For example, because sector 1 is considered first, it is likely to be used frequently at the beginning of the trace. The performance of the workload based upon a descending sort order falls in the middle: Again, examining the same sectors first produces temporal locality; however, because the partial paths are shorter, the distribution of jump distance is not as accurate as the workload based on an ascending order.

Table 5 shows the performance of our modified OLTP workload as compared to the synthetic workloads generated using various sort orders. In this case, the random vertex ordering did lead to the most accurate synthetic workload. We suspect the random vertex ordering was best in this case because, as Table 7 shows, the OLTP workload has a low cache hit rate.

The differences in performance among the different synthetic workloads shows that a single synthetic workload generation technique is not necessarily the best for all workloads. In the case above, we saw how the ascending vertex order produced the most accurate synthetic workload for the E-mail workload whereas the random vertex order was better for the OLTP workload, which has less temporal locality. These results further support the need for a tool like the Distiller, which can accurately evaluate the effectiveness of several different generation techniques.

5. Future work

There are many aspects of our synthetic disk access generator that we have yet to investigate. (1) We plan to further investigate the effects of dividing a workload into smaller parts according to sector number. (2) Our results showed that a random ordering of vertices for the depth-first-search led to the longest partial Hamiltonian paths. We would like to identify precisely what attributes of the random order (as opposed to the ascending or descending vertex orders) lead to the longest paths and possibly define a new, deterministic order that always captures these attributes. (3) Our techniques for generating a complete disk access pattern given a partial Hamiltonian path are somewhat simple. We plan to investigate more complex and accurate techniques.

We also plan to add our generator to the Distiller's library of workload analysis and generation techniques. Once it is a part of the Distiller's library, we can evaluate the usefulness of our jump distance generator when combined with other synthetic workload techniques, such as those that accurately reproduce arrival time patterns, or those that address correlations between I/O request parameters.

6. Conclusions

We have shown that, in a reasonable amount of time, we can generate a synthetic disk access pattern that maintains both a specified distribution of sectors accessed and a specified distribution of jump distance within a reasonable margin of error. Not only is the resulting synthetic workload "cosmeticly" similar to the workload it is intended to model,

but it is also more accurate than synthetic workloads using more traditional generation techniques.

Acknowledgments

This research was begun as part of Hope College's summer REU program, and continued through a grant from the Research and Development office at Grand Valley State University. In addition to our supporters, we wish to thank our colleagues at Hope College for a wonderful summer, Carl Strebel for top-notch system support, and our friends in the Storage Systems Department at HP Labs, who provide encouragement and feedback.

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

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

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

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] Z. Kurmas, K. Keeton, and K. Mackenzie. Iterative distillation of I/O workloads. In *Proceedings of the 11th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2003.

[13] Z. Kurmas, K. Keeton, and K. Mackenzie. Iterative distillation of I/O workloads (extended version). Technical Report GIT-CERCS-03-29, Georgia Institute of Technology, 2003.

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

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

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

[17] 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.

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