

2006

Distributing Jump Distances in a Synthetic Disc Array Workload

Jeremy Zito

Grand Valley State University

Follow this and additional works at: <http://scholarworks.gvsu.edu/cistechlib>

Recommended Citation

Zito, Jeremy, "Distributing Jump Distances in a Synthetic Disc Array Workload" (2006). *Technical Library*. Paper 134.
<http://scholarworks.gvsu.edu/cistechlib/134>

This Poster 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.

Distributing Jump Distances in a Synthetic Disk Array Workload

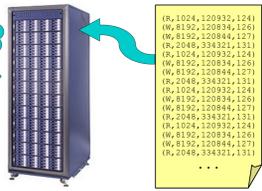
Jeremy Zito

Background

Workloads

Workload traces (lists of individual I/O requests) are often used to evaluate storage systems

Evaluate me!



Where do these traces come from?

Real vs. Synthetic Traces

Real: List of I/O requests made by an application in a production environment.

- Large
- Inflexible
- Hard to find due to security issues

• Perfectly accurate

Synthetic: Generated randomly to maintain high level properties of a workload.

- Compact
- Easily modified
- Contains no specific data

• Rarely accurate

Increasing accuracy of synthetic workloads

Jump Distance

To provide a more accurate synthetic workload, certain properties of an actual trace must be duplicated.

A synthetic trace must share properties that affect how the storage system will behave when serving the workload. One such property of the workload we want to maintain is the distribution of jump distances.



Problem

Jump distance is a representation of the distance the disk head moves between disk accesses. We expect this to affect the storage system's behavior because this physical movement is often a limiting factor in computer systems. To create an accurate synthetic trace, we must maintain a correct distribution of not only jump distances, but also sectors. Generating a list of sectors accessed that maintains both jump distances and sectors used in the target workload, reduces to the NP-complete Hamiltonian Path problem. (Note that jump distance is an approximation of disk head movement, calculated by the difference between successive I/O request starting sectors ($startSector[x] - startSector[x-1]$)).

Our Solution

Our algorithm attempts to solve the Hamiltonian Path problem defined by the jump distance problem by creating a directed graph. One vertex is added for each I/O request. An edge is created between two requests if there exists a jump distance corresponding to the difference between the sectors. We then execute a depth first search beginning at some initial vertex. If we find a path of the desired length, then we have found an ordering of requests that maintains the required jump distance distribution.

Algorithm

Building Paths

We first have to build data structures to hold the jump distance and sector information. We do this by constructing an adjacency matrix:

By searching through the sectors in the original workload, we keep track of every jump distance present and how many times those jumps occur.

For example, the workload to the left would have jump distances of 2, 10, and -40. (sector 32 - sector 30 = jump 2)

The edge is then placed into the adjacency matrix:

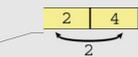
	2	4	30	32	42
2		X			
4					
30					X
32					
42					

Notice that an edge between 30 and 32 was also added as there is a jump of 2 between them.

If jumps of -28, -2, 30, and 10 were also found in the workload, this portion of the matrix would be:

	2	4	30	32	42
2		X		X	
4	X				
30	X			X	
32		X	X		X
42					

We then define all possible edges between sectors based on the jump distances available. For example, if the original workload contained the sectors 2 and 4, we would say there is an edge between them because there is at least one jump distance of 2 (found in the previous step, between sectors 30 and 32).



Each "X" represents an edge between two sectors. Every sector has multiple paths from itself (Xs in each row). For example, sector 2 has a path to 4 and one to 32. Each of these paths are then linked together.

	2	4	30	32	42
2		X		X	
4	X				
30	X			X	
32		X	X		X
42					

Main Loop

We will then search the finished graph (defined by the matrix) for a path that will maintain the sector and jump distance distributions of the original workload.

The Search

Using the mapping of possible paths, we then begin the depth-first search.

• We must first choose a starting sector. For simplicity we use the first sector of the input.

• We then search for the closest possible sector from the starting sector.

Possible paths are 4 and 32. Since the closest of these is 4, it is chosen.

Hiding a sector is simply a matter of removing its links.

For example, if sector 2 was hidden after its first use, it would then be bypassed by anything previously connected to it.

A similar method of hiding is done if the maximum number of jump distances is exceeded (as the distribution of these must be the same as the original workload). If this happens, paths involving the jump are hidden.

Sector 4 would then be set as the current sector. The search would continue by looking for the closest possible path from sector 4. Since there is only one possible path (2) it is chosen.

Synthetic workload: 2 | 4 | 2 | ...

Now our starting sector is once again 2. Remember that the synthetic list must maintain the distribution of sectors present in the original workload. What if sector 2 was only used once in the original workload? To counter this we "hide" sectors that can no longer be used.

It may be the case that as we traverse our map we run into a sector that has no possible paths leading from it. For example, sector 4 no longer has a path to 2 because 2 has been hidden. This leaves no possible paths from sector 4. We then must backtrack through the map and take the closest alternate route.

Finish Randomly

"Finish randomly" is just what it sounds like: the random placement of those sectors "left over" when the search is terminated because it is not making any progress. This random placement of sectors could have an adverse affect on jump distance distribution if the number of "leftover" sectors is large.

Synthetic workload:

546 | 1247 | 443 | 780 | 2140 | 2543 | 1247 | 743 | 2477 | 8345 | 346

Leftovers:

226 | 3460 | 1683 | 360

Final Synthetic Workload:

546 | 1247 | 1683 | 443 | 780 | 2140 | 3460 | 2543 | 226 | 1247 | 743 | 2477 | 360 | 8345 | 346

Finish Up

Because the depth-first search may not finish in a reasonable amount of time, it may be stopped if it fails to make any progress after a set amount of iterations. Approximation techniques are then used to distribute the remaining sectors.

Backtracking

In our example, sector 4 is a dead end:

Synthetic workload: 2 | 4 | ... X

We must then backtrack and chose the next closest path from sector 2.

	2	4	30	32	42
2		X		X	
4	X				
30	X			X	
32		X	X		X
42					

The next closest path is then 32. After adding the new path the search continues:

Synthetic workload: 2 | 32 | ...

While backtracking, we look for possible hidden nodes that may become unhidden. Because backtracking "adds back" sectors and jump distances used, they may not have to be hidden.

In our example, if there had only been one sector 4, it would have been hidden after it was added:

Synthetic workload: 2 | 4 | ...

However, when we backtracked and chose the alternate path to sector 32, sector 4 is "added back" and can then be unhidden because it no longer appears in the synthetic workload.

Synthetic workload: 2 | 32 | ...

* Note that as the synthetic workload is built, we may have to backtrack many times.

Results

The big question is: *How far does the algorithm get?*

As hinted at in the backtracking portion of the algorithm, the length of the path found before the search stalls, is largely determined by the order in which sectors are searched. When we build our matrix, *what happens if we change the order in which sectors are searched?*

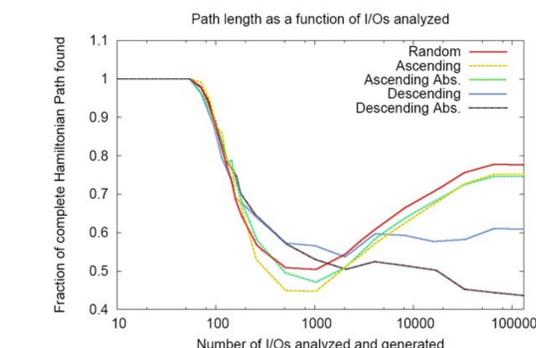
To determine the best way to order our adjacency matrix, we add sectors in one of the following ways:

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

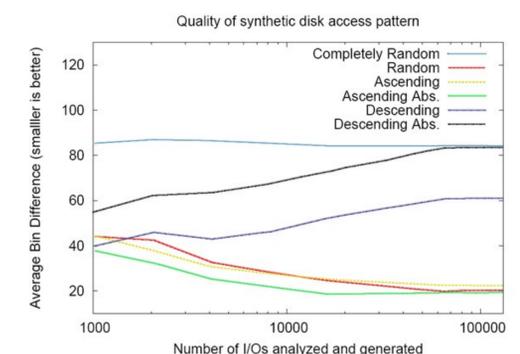
For Example, a random ordering may create the following adjacency matrix:

	2	30	42	32	4
2			X	X	
30	X			X	
42					
32		X	X		X
4	X				

The graphs to the right analyze a synthetic workload, as generated by different sort orders. The synthetic list is based off a "real" workload from an email application.



The longest partial Hamiltonian path is created when using a random ordering of sectors. The next best sorting is ascending. A random ordering will allow us to come up with a different solution every time the algorithm is run; unlike the others, which will all give the same ordering and thus, the same results. Also, notice that a complete path is found only for workloads with less than 100 I/Os. However, most workloads are very large, containing thousands and thousands of I/Os.



One metric used to determine the quality of a synthetic workload is "average bin difference." This is simply the difference between the number of jump distances used in the "real" workload and those used in the synthetic workload. For example, a jump distance of 2 may be found 20 times in the "real" workload, but only 12 times in the synthetic workload, a difference of 8. This difference is calculated for every jump distance, and the average difference is then calculated. Once again we see that the random and ascending search orders produce more useful access patterns.