HPL–SSP–2001–4: Simple table-based modeling of storage devices

Eric Anderson *

July 14, 2001

Abstract

Trace driven simulations are too slow for use in solvers. Analytic models require work from a person to understand the array enough to model it. Table based models offer the possibility of automatically measuring the performance of an array for use in a solver. We explain a simplistic way of generating the input points in the table. We then explore three different ways of performing the interpolation of nearby points from the points within the table, and comment on future directions the work could go.

1 Introduction

Traditional approaches to building disk array models involve either slow trace-driven simulation, or analytic models [UAM01, MA01] which rely on understanding the underlying behaviour of the disk array. Both of these approaches mean that creating the model for a new array, or even just an updated model for a new revision of an array is a human-intensive process. We propose instead an almost entirely automatic method which requires limited understanding of the disk array, yet still gets performance predictions of the same quality as analytic models. These analytic models are useful for performing fast searches through many different possible configurations [AKS+01].

Our methodology consists of two parts, first a measurement technique for sampling the different possible performance values of the array, and then a algorithm for taking the resulting performance table and estimating the performance for arbitrary inputs.

The resulting model takes as input a series of parameters, and returns the maximum estimated throughput available for those parameters. Some of the parameters specify the device information. For our models these parameters are the raid level, and the number of disks in the raid group. The second set of parameters are a summary of the request pattern, or stream as done in [ABG+01]. For our models, these parameters are the request type (read or write), the request size, the sequentiality of the requests, and the average queue length.

*Storage and Content Distribution Department, Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 95014; email: anderse@hpl.hp.com.

2 Building the performance table

For the purposes of this initial tech-note, we use the simple grid approach to building the performance tables. In this approach, for each of the parameters we specify the possible values for the parameter, and measure the performance at those parameter points. A better approach would be to do additional sampling at those places where the estimated performance is changing quickly, and/or where the performance estimation is greatly off.

Taking these measurements is straightforward. We configure the array to have LUs of all the different types we wish to measure, and then for all of the stream parameters, we measure the maximum performance that can be achieved for those parameters by issuing I/Os as quickly as allowed by the parameters (limited by the queue length and the completion rate of the device). We chose to simply take measurements for a fixed amount of time because that is what our synthetic I/O generator supported. A better approach would be to take measurements until the confidence interval for the measurement becomes sufficiently small.

3 Estimating the performance

Given a table of measured values, there are many ways of estimating the performance at an arbitrary point. We chose first to partition the table based on parameters we did not expect would be usefuly predictive. In our case, that meant creating separate tables for each raid level, number of disks in the raid group, and each operation type. This choice is ok because we have an exhaustive list of all possible values for those parameters, so we will not have to interpolate between them. We leave to fu-

ture work the question of whether interpolations between these parameters can be done.

Given a table restricted to the relevant measurements, we must now estimate the measurement at points not in the table. Assume we have a point $p := (p_1, p_2, p_3) \in R^3$ representing the request size, sequentiality and queue length. We first start by restricting p to the ranges that we have measured. For example, if the range on the queue length is $[8, 64]$, we set p_3 to 8 if it is less than 8, and 64 if it is greater than 64. We can now more safely interpolate between the points in the table for our estimated value. All of our algorithms use a notion of closeness. To get the distance, we took the euclidean distance after normalizing all of the points to be in the range $[0, 1]$. The normalization was necessary because some parameters have a wide range, and some have a narrow range, but we wanted the contribution from each axis to be the same.

We have explored three methods for doing the interpolation.

- **Closest point**. Given the euclidian distance between p and the each point in the table, assume the maximum throughput at p is the maximum throughput at the closest point.

- **Nearest neighbor averaging**. Chose the k nearest neighbors. Weight the contribution of each neighbor by it's distance from p such that the sum of the weights add up to one. Estimate the utilization at p as the weight of each neighbor times the utilization at that neighbor.

- **Hyperplane interpolation**. Chose nearby points which form a non-degenerate hyperplane. Estimate the performance at p as the value of the hyperplane at p.

We found that the closest point algorithm worked surprisingly well. The nearest neighbor averaging algorithm had a fatal flaw which rendered it unworkable for our set of measurements. The hyperplane interpolation was tricky to get working, but in the end performed better than the closest point interpolation. We believe in the future that trying spline interpolations is likely to work even better than hyperplane interpolation.

For our simplistic evaluation, we took a random fraction of the table, and used that random fraction to predict the values which were not in the table. We use that metric to estimate the quality of the estimation algorithm. A better approach of course would be to sample at additional points, but we did not have the time to take additional measurements.

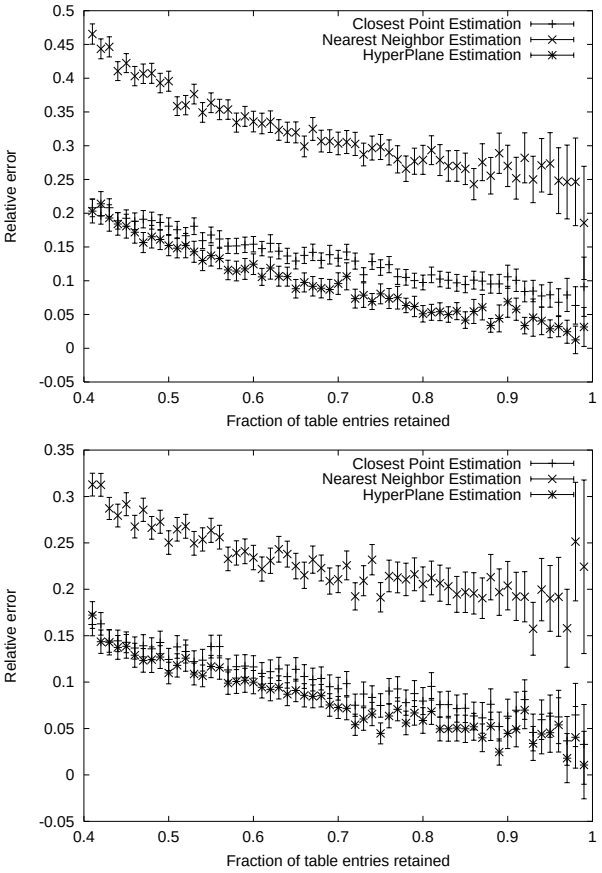

Figure 1: Only a fraction of the measured points are used to estimate the performance at the excluded points. 50 measurements were taken at each point. Error bars are the 95% confidence intervals. Top graph is from the RAID-1 measurement data, and the bottom graph is the RAID-5 measurement data. Nearest neighbor estimation is clearly the worst. Hyperplane estimation does somewhat better on the RAID-1 table, and slightly better on the RAID-5 table.

3.1 Closest point

The closest point algorithm is convinient in its simplicity. It also works surprisingly well. Figure 1 shows how the quality of the estimation drops off as the fraction of the retained table is dropped.

3.2 Nearest neighbor averaging

Nearest neighbor averaging works poorly to estimate the performance. It's fatal flaw comes at estimating values near the edge of the measured space. Figure 2 shows this problem graphically. If the data tends to be sloping in one direction from the edge, then any system which takes a group of points near the edge and averages them together

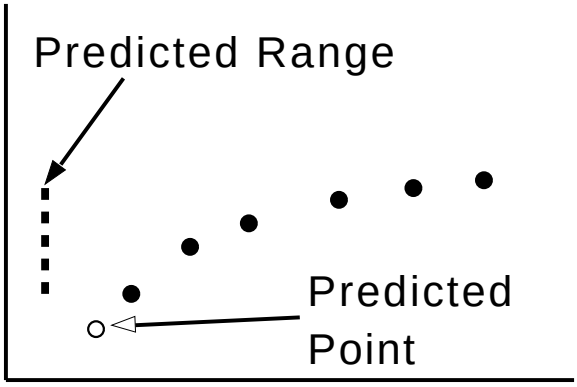

Figure 2: Illustration of the problem with nearest neighbor averaging. The point being predicted (open circle) is in the lower left, and all the nearby points found in the table (filled circles) are up and to the right. Any possible averaging of the points in the table will end up somewhere in the range indicated on the far left, well outside of the actual y value of the predicted point

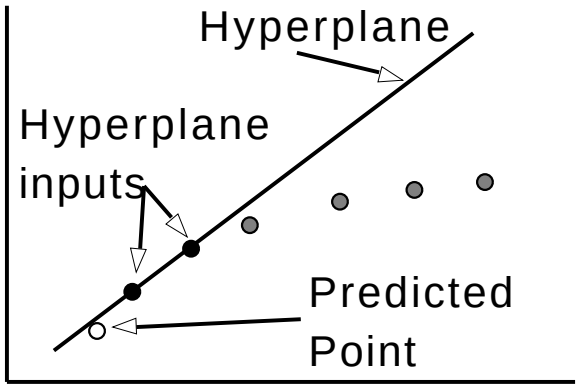

Figure 3: Illustration of hyperplane interpolation. First the algorithm selects 1 nearby point (black ones) per dimension out of the complete set (gray points). Then the hyperplane which passes through all of those points is calculated. Finally, the prediction is done as the point on the plane above the input point. In this example, the prediction would be slightly too high.

(by whatever means) will get fairly poor results.

In our implementation, we chose the nearest $n := 6$ points. The distance to each neighbor from the point being estimated is d_i. We then calculated the non-normalized contribution(c_i) to each neighbor as $c_i := 2 * \texttt{max}_{j \in 1..n}(d_j) - d_i$. The normalized contribution for each neighbor is then $c_i / \texttt{sum}_{j \in 1..n}$. Figure 1 shows how the quality of the estimation is worse than for the closest point algorithm.

3.3 Hyperplane estimation

Our final, and best algorithm of the ones we implemented was hyperplane estimation. The idea is to choose a set of nearby points, calculate the hyperplane which runs through them, and use that hyperplane to estimate the maximum performance at the target point. Figure 3 shows how this works for an interpolation in two dimensions.

The biggest problem with this algorithm was avoiding degenerate hyperplanes. Given the input data we had, it was very common when estimating the performance of a large I/O, say 240k, that all of the nearest points would be ones at 256k. As a result, the hyperplane would be degenerate.

Our first attempt at avoiding degenerate hyperplanes was to keep track for each axis, the point which is nearest and on that axis, nearest and smaller on that axis, and nearest and larger on that axis. Unfortunately, this approach lead to a high duplication of points in the final set. Consider the case where you are estimating the value at $(1.1, 1.1, 1.1)$ and you have nearby points at $(1,1,1)$ and $(1.5, 1.5, 1.5)$. It is likely that those two entries will occupy most of the closest set, again leading to a degenerate matrix.

Our second approach was to extend the first approach such that if a point had been selected for an earlier axis, it was not checked for the later ones. The one disadvantage of this approach is that in our single-pass algorithm, the strictly nearest point might not be retained for all of the entries in the set. Consider the case where the first point is the overall best on the third axis. It will get selected as the nearest for the first axis, and never considered for the third. However, as the scan over the table continues, it will be overwritten as the nearest point on the first axis. There are assuredly more complex algorithms which could fix this problem, we did not investigate them.

As a final addition to our second approach, we also retained the strictly nearest point.

Given the set of points, the next problem is to choose the hyperplane through them. One approach would be to do a least squares fit to the points. We instead chose to do a perfect fit to the nearest points, assuming that those would best describe the shape of the hyperplane. For this purpose, we sorted the nearest points in order of their distance to the target point. We then started iteratively adding them to a hyperplane matrix. Any point which was redundant with the earlier points was eliminated. We tested redundancy using gaussian elimination. Eventually we would either get to 4 entries in our matrix, which is sufficient to define a hyperplane in a 4-dimensional space (size, queue length, run-count, measured performance), or we would give up. We also retained the minimum and

maximum performance seen at any used point.

If we failed to generate a hyperplane, then we simply fell back to our closest point algorithm. If we generated a hyperplane, then we used it to estimate the performance. As a final sanity check, we forced the estimated performance to be bounded by $[1/2 * min\_used, 2 * max\_used]$.

Figure 1 shows the performance of the hyperplane estimation is better than the performance of the other two algorithms as the number of points excluded increases.

3.4 Combining together multiple measurements

For the purpose of a solver, we need to be able to estimate the utilization of a device with multiple simultaneous accesses to it. However, we know that two different access streams will interfere with each other. In particular, having multiple streams will increase the average queue length and decrease the average sequentiality. We used the same algorithms as in [UAM01] to calculate the inter-stream adjustments. We then estimated the utilization for each stream as the rate of that stream divided by the maximum at the adjusted point. Finally we combined the inter-stream estimates using the algorithms found in [BGJ+98] to handle inter-stream phasing.

4 Conclusion

We have shown that the hyperplane interpolation algorithm performs better than the closest point algorithm and the nearest neighbor averaging algorithm. We have shown that regardless of the data, the nearest neighbor algorithm is unlikely to perform well. In the future, we would like to refine our methodology for choosing the points to measure. In particular, we would like to take more measurements in the regions where the performance changes rapidly and fewer where it does not. We would also like to change the hyperplane interpolation to be spline interpolation. We believe splines will work well as they do not give wildly varying predictions the same way high-dimensional polynomials do. We would also like to find a faster way of selecting the nearest points. We suspect that the R-tree algorithms [Gut84] may form a basis for this, although the implementation may be somewhat different as we have an entirely in-memory database.

References

[ABG+01] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: an automated resource provisioning tool for large-scale storage systems. Technical Report HPL-2001-139, Hewlett-Packard Labs, June 2001. To appear in ACM Transactions on Computer Systems.

[AKS+01] E. Anderson, M. Kallahalla, S. Spence, R. Swaminathan, and Q. Wang. Ergastulum: An approach to solving the workload and device configuration problem. Tech. rep. XXX-YYY, Hewlett-Packard Labs, June 2001.

[BGJ+98] Elizabeth Borowsky, Richard Golding, Patricia Jacobson, Arif Merchant, Louis Schreier, Mirjana Spasojevic, and John Wilkes. Capacity planning with phased workloads. In *Proceedings of the First Workshop on Software and Performance (WOSP'98)*, pages 199–207, Oct 1998.

[Gut84] A. Guttman. trees: A dynamic index structure for spatial searching, 1984.

[MA01] A. Merchant and G. A. Alvarez. Disk array models in Minerva. Technical Report HPL-2001-118, Hewlett-Packard Labs, April 2001.

[UAM01] M. Uysal, G. A. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. In *Proceedings of Ninth MASCOTS*, August 2001.