Towards an Autonomic Storage System to Improve Parallel I/O [*]

Francisco Hidrobo[1,2]
[1]Universidad de Los Andes, Venezuela
email: fhidrobo@ac.upc.es

Toni Cortes
[2]Universitat Politècnica de Catalunya, Spain
email: toni@ac.upc.es

ABSTRACT
In this paper, we present a mechanism able to predict the performance a given workload will achieve when running on a given storage device. This mechanism is composed by two modules. The first one is able to learn the behavior of a workload in order to be able to reproduce its behavior later on, without a new execution, even when the storage drives or data placement are modified. The second module is a drive modeler that is able to learn how a storage drive works in an automatic way, just executing some synthetic tests. Once we have the workload and drive models, we can predict how well the application will perform on the selected storage device or devices or when the data placement is modified. The results presented in this paper will show that this prediction system achieves errors below 10% when compared to the real performance obtained. It is important to notice that the two modules will treat both the application and the storage device as black boxes and will need no previous information about them.

KEY WORDS
Autonomic storage system, Parallel I/O, Disk drive modeling, I/O workload modeling, performance prediction.

1 Introduction

There are many applications where the I/O performance is a bottleneck and thus many solutions have been proposed. One of the most promising ones consists of configuring the storage system and data placement to maximize the storage-system performance for a specific workload. In general, this approach consists of finding the optimal configuration and data placement for the I/O system given a specific application or set of applications and a set of storage devices. Currently, these optimizations are done by experts who use their experience and intuition to make this configuration and placement. This means that only a few sites can take advantage form this kind of "optimal" placement benefits because not everybody has (or can afford) an expert to place data in the best possible way. For this reason, a tool that could perform this tuning in an automatic way would be a great step in making this technique available to a wider range of sites. Furthermore, this tool becomes even more useful if the optimal configuration and placement varies throughout the time making it more difficult to keep the right placement up to date.

Our mid-term objective is to design a storage system capable of extracting all potential performance and capacity available in a heterogeneous environment with as little human interaction as possible. We envision the system as an advanced data-placement mechanism that analyzes the workload to decide the best distribution of data among all available storage devices, as well as the best placement within each device. Figure 1 presents the system architecture with its modules and the relationship between them.

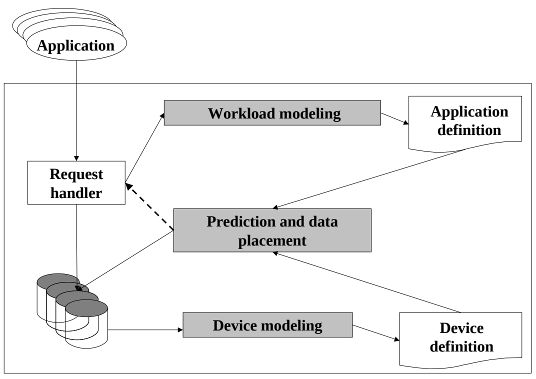

Figure 1. Block architecture for our system

The objective of this paper is to describe the two first modules: *device-modeling* (from now on, we will use drive and disk indistinctly, although our model will be able to learn the behavior of other storage devices than disks) and *workload-modeling module*. In addition, we want to verify the effectiveness of the prediction that is achieved by these models working together.

We will present a model for disk drives based on Neural Networks. This model was first proposed in a previous work by the same research group [8], but in this paper we want to show that it adapts well to the model we propose for the workload. It is important that both models work well together or the final results will lie behind the expectations. Regarding the model for the workload, we will present an approach based on files (not on application) that will allow us to concentrate the effort in the most important files instead of on global storage data. In addition, this model will be able to reproduce the behavior of the workload when a new data placement is used without loosing accuracy (and without having to rerun it).

We tested our approach with four benchmarks: three

[*]This work was supported in part by a grant from FONACIT (Venezuela) which is gratefully acknowledged, by the Ministry of Science and Technology (Spain), and by FEDER funds of the European Union under grants TIC2001-0995-C02-01.

synthetic applications and the well known TPC-H. For each case, we built the file model and verified that this model along with the drive model actually reproduce the real performance with an error below 10%.

2 Autonomic storage system: a global picture

When the system is initially started, or when a new storage device is added, the model of the new drive or drives is built. The objective of this model is to be able to give an estimation of the performance a given workload will achieve for a specific data placement. To build the drive model, we plan to execute some synthetic tests, trace the behavior of the disk, and use these traces to train the model (always taking the drives as a black boxes). Fortunately, this model will not change during the live of the system because it is not application dependent, and thus the training time is not a critical issue.

Once we have all storage devices modeled, we run the applications normally on top of it. During this normal execution, the workload modeler keeps track of the accesses done to all the devices and it builds a model of what is happening in the system. This model should be good enough to be able to regenerate a trace of an execution with the same characteristics than the real execution.

Once we have gathered enough information about the accesses to the storage system to have a clear picture of what is going on, we start studying what are the important files/blocks, access patterns, etc. With this information, the data-placement module decides which possible changes make sense to improve the performance of the storage system (i.e. decide which data goes to which disks, how is data stripped, interleave related files, put special blocks in the fastest disks, or faster zones in the same disk, etc.) Hopefully, we will have many different data placements that may improve the actual distribution. Then, we feed the workload information and the new placements into the drive model to decide the performance each new placement would achieve. Afterwards, we compare all possibilities and decide which one is better and whether the new placement is worth the effort of making the changes.

3 Disk Model

The main objective of this module, and also the main difference compared to other approaches, is to design a model that has no previous knowledge about the drive to model. This will allow us to use mathematical tools to implement it and, we will be able to apply this approach over a wide variety of storage systems with minimal (hopefully none) additional effort.

Although this model was already proposed in a previous work [8], we will give the most important detail about it, because we need to show that this model works well with our workload model and because it has been tested on global traces, but never on per-file basis, as we will have to do from now on.

3.1 Model approach

We propose a general model based on a mathematical function. We assume that we can find a function (M) that approximates the service time St for each request. Thus, our general model can be expressed by:

$$St \approx M(R)$$

where: St is the request service time and R is the input vector with components:

- $R.Addr$: address of the first-requested block,
- $R.Jump$: difference (in blocks) between this request and the previous one. We do not use the difference in cylinders because that would mean a knowledge about the internal design of the drive.
- $R.Size$: request size.

Our experience has shown us that it is better to have one model for each operation (read and write). For this reason, we will use one model for each request type.

Once built the model, it can be used to find the response time for each request done by any application, and from there compute the throughput.

In [8], we studied different approaches and we found that neural network are the best mechanism to model drive behavior. This study took different applications and storage devices into account (including disk drives and other devices such as memory disks).

3.2 Model based on Neural Network

Neural Networks have a high capacity for function approximation [7], and this is exactly our objective. The neural network architecture we have used, derived from our previous experiments, is a feed-forward neural net with following configuration:

- 3 neurons in input layer (R.Addr, R.Jump and R.Size).
- 25 neurons in the hidden layer using hyperbolic tangent sigmoid transfer function.
- one neuron in output layer (service time) using linear transfer function.

To resolve the problem, we use a Levenberg-Marquardt back-propagation algorithm [12].

In [8], we showed how the neural network should be trained, but essentially a synthetic application is executed using the device to model. Then, the resulting trace is used to train the neural network.

4 Workload model

After studying different possibilities, we have decided to focus the workload model on files instead of on applications, processes, or any other possibility. We envision the system as an advanced data-placement mechanism, and these data is associated to files. Furthermore, this file modeling will allow us to focus the effort towards the most important files of the workload to reduce the size of model and the complexity of data-placement module.

We can establish a criterion to order the files in accordance to their importance and apply the mechanisms of redistribution to the most important files. This order criterion can be based on the percentage of the global number of access or in the percentage time employed for I/O operations on the file.

As we want a file model to be used with our drive model, we need to learn it from as close to the disk as possible. For this reason, we model the behaviour just before the disk drive (after the filesystem cache and most of the operating system interaction). For this study, we have modified the linux kernel to save this information in the point where the blocks are requested to the disk drive, and this where the integrated version should also extract its information.

4.1 File model approach

We will have a file model per each file that fall in the "important" category according to their usage.

The first idea to model a workload is to use a trace, unfortunately we cannot use it because it is not flexible enough to adapt to the behavior of a new storage drive or a new placement of data because traces are device and placement dependent. In addition, a traditional trace grows for ever if the application repeats its behavior once and again. This repetition will appear for sure if the workload is to be predictable because we can only predict what has already occurred at least once.

The intuitive idea of our model is to have the advantages of a trace file such as accurate representation of the workload, detailed description of accessed blocks and their relationship but without the disadvantages mentioned above.

Our file model is based on the segments accessed and their relationship. We define a segment as a set of blocs requested in a single disk operation. Please keep in mind that our model works at disk-drive level and thus segments are not necessarily what users request, but what the operating system (or filesystem) sends to the disk drive after being filterer by the buffer cache and after some possible reordenation done during the process. Thus, our file model stores, for each segment, the following information:

- The first logical block requested. Using the logical blocks makes our model independent of the physical location of the block. Although we model at very low level, we do not loose the concept of file. In order to limit the size of the model, all segments that begin in same block will only use one entry in the model.
- The total number of requests made to this direction. In order to keep some information about the importance of different segments, we maintain a counter on the number of times a segment is requested.
- Request type (read/write). This value is expressed as read probability. Remember that this model will be used to build a trace for our disk drive and thus, with a probability will be enough to build this trace. In addition it saves space because we do not need to keep all the different requests to the same segment.
- The average size of the requests made to this segment. Once again, this is enough and reduces the size of the model significantly compared to keeping all requested sizes. We will show that the error this simplification introduces is affordable.
- The list of possible predecessor requests. This list contains information about the requests that preceded it in the trace. Remember that the trace used by the disk drive model needs to know the jump (in blocks) done by the disk from the previous request to the current one (used to model the head movement in case the drive has it). Each entry in the list has the following information:
  - the file id. This field is used for two reasons. The first one is to identify the physical location of the last accessed block, without keeping the physical location in the trace. Second, it is used to find relationships among different files. Note that if a modeled file has relation with another file, which is not modeled, the relationship is modeled by this field.
  - the last logical block of the previous request. This field is used to find the number of blocks the disk has "jumped" from one request to the other. This is necessary for us to build the trace needed by the disk model.
  - the number of times that this element preceded the request. This field is used to know the percentage of times this sequence occurred. Once again this reduces the size of the model when a the same sequence of requests is repeated.

This model fulfills all our requirements. It can be generated on-line because new requests can be added very easily. No physical information is kept, and thus it is block-location independent. It does not keep any time information and thus it is device independent. It does not grow unnecessarily but it has enough information (even inter-file relationships) for the data placement module to take decisions. Finally, building a trace with the information required by the drive model previously described (R/W, initial block, jump and size of the request) is trivial.

5 Methodology

In [8], we tested the disk model with three real workloads. However, in those tests we did not use file separation and the whole trace was fed into the drive model to test its accuracy. Working at file level is different because less requests can be used and thus the results may differ. In this paper, we are going to show that our disk model approach is adequate to predict performance for specific files. Also, we will show that the file model allows us to estimate the I/O performance and that it can be used to remake the trace for a different drives or data placements with good accuracy.

5.1 Workloads

5.1.1 Synthetic benchmarks

We have defined three synthetic benchmarks that will help us observe the behavior of our predictor in a "controlled" environment:

1. Simple Sequential Access (SSA). The application reads one file sequentially. In each read it requests a variable size between 8 and 512 KB. The inter-arrival time is uniformly distributed between 0 and 100 ms. It will show us the behavior in the simplest case: accessing a single file sequentially (but no necessarily using regular sizes or intervals).

2. Multiple Sequential Access (MSA). Here, the application reads three files simultaneously. It requests a segment of the first file, then a segment of second one and a segment of third file. These operations are repeated until all bytes have been read. The sizes distribution and inter-arrival time are the same as the ones used in SSA. This benchmark will allow us to observe what happens when several files are accessed concurrently, but with a perfectly reproducible behavior.

3. Concurrent Sequential Access (CSA). Three instances of the SSA applications are run in parallel (each using a different file). The behavior will be similar than in the previous benchmark, but without the reproducibility characteristic (because the operating system and the other applications may affect differently in each execution).

The files used were 300Mbytes each and we called them: $file_1$, $file_2$ and $file_3$.

5.1.2 TPC-H benchmark

In order to test our system using a real application, we have decided to use the TPC-H benchmark. It is a decision support benchmark for database. It consists of a suite of business oriented ad-hoc queries and concurrent data modifications. It has been development by the Transaction Processing Performance Council (TPC) [17].

We created a database of 1 Gbytes and used *Postgres SQL 7.2.1* for LINUX. We ran all queries except queries number 7, 9, 20 and 21 because these queries contain not supported functions in our database manager system.

For this benchmark, we took the files where the application spent over 90% of I/O time. These files are three and they represent the tables *partsupp*, *order* and *lineitem* in the database.

5.2 Experimental Results

5.2.1 Prediction of the modeled workload

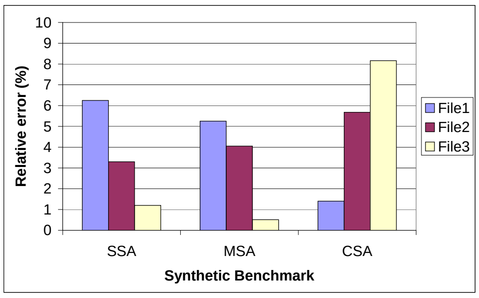

Figure 2. Error for performance prediction in synthetic workloads

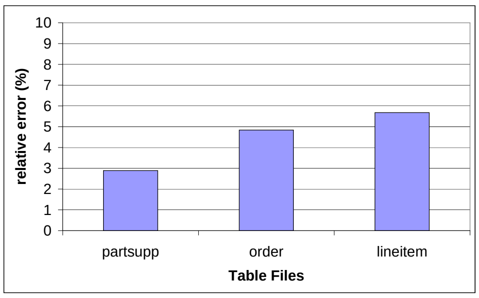

Figure 3. Error for performance prediction in TPC-H

This first experiment tries to show that the workload, along with the drive model, can predict the performance achieved by the real execution used to model the workload.

Figure 2 shows the results for the three synthetic workloads. Here, we observe that the error obtained by the model is always below 10% in all cases. Similarly, for TPC-H), the error obtained for all files is also lower than 10%.

These results show that the proposed models work properly and that the workload granularity is good enough to make an accurate prediction.

5.2.2 Prediction of the modeled workload using a modified data placement

The objective of this second experiment is verify if our application models can be used to predict the performance when block distribution have changed, which should be a very frequent situation. To test it we randomly moved 10% of the blocks in a selected file and we executed all the tests again. In the synthetic benchmarks we selected $file_3$ to move 10% of its disk blocks whereas in TPC-H we selected *lineitem* because is the biggest file in the application.

We took the file model obtained with the block distribution where blocks were moved randomly and generated an estimate trace for the new block distribution. We should notice, that although the workload model is device and placement independent, it has to be learnt from a real execution. In this experiment we show that it is really independent and that the model learnt form a placement can be used with no problem to predict another.

Figure 4 shows that our system is able to make a good performance prediction with errors below 10 %.The same experiment was done with the TPC-H benchmark and the error observed was 3.18% for file *lineitem*.

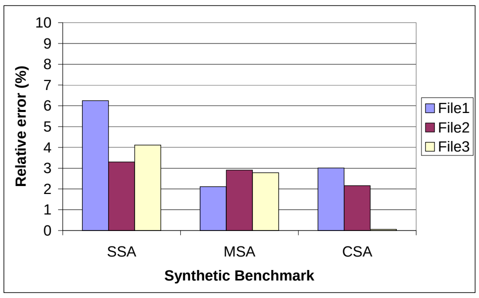

Figure 4. Relative error for synthetic workloads with block redistributions

5.2.3 Prediction of the modeled workload when the data is changed

We also wanted to study what happens when the changes in the block distribution and/or access pattern are produced by the application itself, and not by the data placement. For this reason, we ran the two refresh functions specified in the TPC-H benchmark. After each execution of refresh functions, we measured the throughput obtained using file model and the throughput obtained by the original database.

Figure 5 presents the results for this test. In the x axes we represent the percentage of modification done by the refresh function (from 0% to 10%, and where 0% represents the original database). We should note, that the refresh functions, as are specified by the benchmark, should only change 0.1%, and thus our modifications are huge ones compared to the expected ones. Nevertheless, it can give us an idea of how sensible our model is to this kind of changes.

First, we can observe that the error between the model and the real execution for the *order* file grows as the percentage of refreshment also grows. This growth occurs because this is the file where records are added and removed, and thus the pattern changes with each refresh function. The valley observed is because the error goes from negative to positive, but the important conclusion is that the model can handle some modification to the original file. We should keep in mind that these changes will not be frequent in our target environment. In addition, if the file changes sufficiently, we just need to model it again to have accurate predictions.

Second, *lineitem* is also one of the modified file but we cannot observe any change. The reason behind this different behavior is that Postgres saved the updates for this table in a different file and thus *lineitem* was not really modified (this policy is implemented for the large tables).

Finally, the *partsupp* file is not directly affected by the refresh functions, and thus its behavior is not modified.

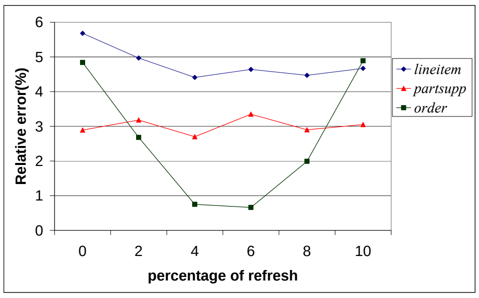

Figure 5. Model error with refresh functions

5.2.4 Size of file model

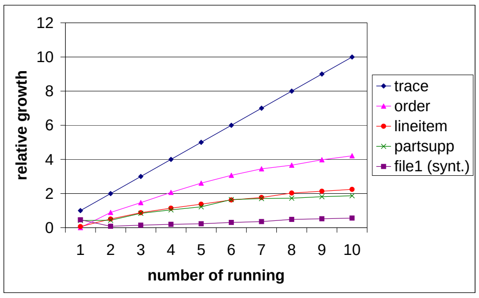

Figure 6. Growth of model size

In order for a workload to be predictable, it has to repeat its behavior from one run to another (or at least, have a similar one). One of the objectives of our model, was to keep all necessary information about the workload, but not to repeat already observed information. For this reason, we want to see if our model grows after several repetitions of the same workload. In order to test it, we repeated the tracing process 10 times for synthetic benchmark CSA with $file_1$ and took the size model obtained in the TPCH benchmark with the refresh function. This experiment allows us to compare the size of the model with the size that would have the accumulated trace.

For each execution we took the increment in the size of file model with regard to the original size of the trace after the first execution. Figure 6 shows these curves for files *partsupp*, *order* and *lineitem* from TPCH benchmark, and $file_1$ from synthetic benchmark SSA. This figure also contains a curve for the growth of the trace.

We can observe that the file size for $file_1$ in synthetic benchmark is maintained almost constant because this file is not modified. The growth of file model is due to filesystem behavior. However, in TPCH benchmark, the file model for *order* grows because this file is being modified by the refresh functions in each run. Nevertheless, the file model size always grows significantly less than the trace size.

6 Related Work

In the area of storage-drive models we can find several approaches such as simulators [13, 4] that can even get their parameters automatically [14]. Another possibility is the usage of analytical models where the input model is not a trace, but a characterization of the load [15, 18]. In both approaches, the disk drives are not treated as a black box, which means that they need of plenty of support if we want to use then in a changing heterogeneous system. Our neural-network model does not need to know anything about the device and it can learn it automatically. In the same area, there is also a group of proposals that treat the drive as a black box and learn the behavior after a training period [16, 2]. The problem with these approaches is that they are not fine grained enough or they need too much time to predict the performance.

With regard to application models, some work have been done in characterizations and modeling of I/O access pattern [9, 10, 11, 19, 20]. Most of these studies were made at the filesystem level and not at disk level, as we need. Ganger in [5] made an interesting contribution in characterization of storage workloads and workload synthesis, but he was unable to accurately recreate the access and arrive pattern of the traces and he showed that commonly-used simplifying assumptions about workload characteristics are inappropriate and can lead to erroneous conclusions. Based on Ganger's work, Gómez and Santoja [6] developed a approach to analyze and modeling disk access pattern base on self-similarity concepts. Using this approach is not possible to associate the performance to some specific file directly, it can not be used to predict how the blocks redistributions change the operation either.

Regarding our global system, there are some tools similar to our system such as MINERVA [1] and Hippodrome [3] development in the HP Laboratories. The structure of their systems is similar to our architecture. However, they target their work to the configuration of RAID systems, whereas our system uses learning to improve the performance by means of the data redistribution.

7 Conclusions

In this paper, we have shown the effectiveness of our performance prediction system based on a storage drive model and a file model (both learned automatically). We verified that our predictions achieve errors bellow 10% even if the data placement is modified, the application behavior changes slightly or the data used is somewhat changed. Thus, we have shown that our proposal is exactly what we need to build the autonomic storage system that places data in the most efficient way in a heterogeneous storage system.

Finally, we want to point out that the results presented in this paper are significant enough without the whole system because they can be applied in different scenarios than the one we have in mind.

References

[1] ALVAREZ, G., BOROWSKY, E., GO, S., ROMER, T., BECKER-SZENDY, R., GOLDING, R., MERCHANT, A., SPASOJEVIC, M., VEITCH, A., AND WILKES, J. *MINERVA: an automated resource provisioning tool for large-scale storage systems*. *TOCS 19*, 4 (Nov 2001), 483–518.

[2] ANDERSON, E. Simple table-based modeling of storage devices. Technical Report HPL-SSP-2001-04, HP Laboratories, Jul 2001. http://www.hpl.hp.com/SSP/papers/.

[3] ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. Hippodrome: running circles around storage administration. In *FAST'02* (January 2002), USENIX, Berkeley, CA., pp. 175–188.

[4] GANGER, G., WORTHINGTON, B., AND PATT, Y. The DiskSim Simulation Environment (Version 2.0). http://www.ece.cmu.edu/˜ganger/disksim/.

[5] GANGER, G. R. Generating Representative Synthetic Workloads. An Unsolved Problem. In *CMG Conference* (Dec. 1995), pp. 1263–1269.

[6] GÓMEZ, M. E., AND SANTOJA, V. A new approach in the analysis and modeling of disk access pattern. In *ISPASS* (Austin, Texas, Apr. 2000), IEEE Press, pp. 172–177.

[7] HASSOUN, M. H. *Fundamentals of Artificial Neural Networks*. MIT Press, Cambridge, 1995.

[8] HIDROBO, F., AND CORTES, T. Towards a zero-knowledge model for disk drives. In *ACM'03* (Seattle, WA, June 2003), IEEE Computer Society Press.

[9] KOTZ, D., AND NIEUWEJAAR, N. Dynamic File-Access Characteristics of a Production Parallel Scientific Workload. In *Supercomputing'94* (Washington, DC, Nov. 1994), IEEE Computer Society Press, pp. 640–649.

[10] KROEGER, T. M., AND LONG, D. D. Predicting File System Actions from Prior Events. In *USENIX'96* (California, Jan. 1996), Usenix Associations.

[11] KROEGER, T. M., AND LONG, D. D. The Case for Efficient File Access Pattern Modeling. In *HotOS-VII* (Arizona, Mar. 1999), IEEE Computer Society.

[12] MARQUARDT, D. W. An Algorithms for the Least-Squares Estimation of Nonlinear Parameters. *SIAM Jounal of Applied Mathematics 11*, 2 (June 1963), 431–441.

[13] RUEMMLER, C., AND WILKES, J. An introduction to disk drive modeling. *IEEE Computer 27*, 3 (1994), 17–28.

[14] SCHINDLER, J., AND GANGER, G. R. Automated Disk Drive Characterization. In *Sigmetrics'00* (June 2000), ACM Press, pp. 109–126.

[15] SHRIVER, E., MERCHANT, A., AND WILKES, J. An analytic behavior model for disk drives with readahead caches and requests reordering. In *SIGMETRICS'98* (Madison, Wisconsin, June 1998), ACM Press, pp. 182–191.

[16] THORNOCK, N. C., TU, X.-H., AND KELLY FLANAGAN, J. A STOCHASTIC DISK I/O SIMULATION TECHNIQUE. In *WSC'97* (Atlanta, GA, USA, Dec. 1997), ACM Press, pp. 1079–1086.

[17] TRANSACTION PROCESSING PERFORMANCE COUNCIL. TPC-H. http://www.tpc.org/tpch/.

[18] UYSAL, M., ALVAREZ, G. A., AND MERCHANT, A. A Modular, Analytical throghput Model for Modern Disk Array. In *MASCOTS* (Cincinnati, Ohio, Aug. 2001).

[19] WARE, P. P., JR., T. W. P., AND NELSON, B. L. Modeling File-system Input Traces via a Two-level Arrival Process. In *WSC* (1996), pp. 1230–1237.

[20] WARE, P. P., AND THOMAS W. PAGE, J. Automatic Modeling of File System Workloads Using Two-Level Arrival Processes. *ACM TOMACS 8*, 3 (July 1998), 305–330.