SynergyFS: Artificial Intelligence to optimize performance of Hybrid Hard Disks

Ammar Husain
General Engineering Department
University of Illinois
Champaign, Illinois
ahusain4@illinois.edu

Kartikeya Dubey
Computer Science Department
University of Illinois
Champaign, Illinois
kdubey2@illinois.edu

***Abstract*— Hybrid storage architecture is an efficient method that can optimize I/O performance, cost and power consumption of storage systems. Currently data management is controlled by the users. The system users decide whether to place the data on the solid state disk or the hybrid hard disk manually.**

This paper presents an artificial intelligence mechanism that optimizes I/O performance based on the user's trace data. The mechanism is placed at the operating system level and possesses the ability to move files between the solid state disk and the hard disk drive. For example, by looking at the processes that the user has been running and the files being utilized by the processes the algorithm will automatically move files from the hard disk drive to the solid state disk to reduce access times for files that have a high probability of being used in the future. The mechanism takes into account several parameters in order to make a decision about the migration of files.

Hybrid hard drives; artificial intelligence; machine learning; I/O optimization; access time;

I. INTRODUCTION

A hybrid hard disk is a combination of a 2-6 Gigabyte solid state drive and 250-320 Gigabyte hard disk drive. Hybrid hard disks offer a cost effective solution to users looking for an alternative to hard disk drives yet not being able to afford large capacity solid state drives. However, these hybrid hard disks are not being used to their maximum potential as the user decides where data is to be placed manually. Since the user does not possess an intricate knowledge about the files being accessed by processes that are currently running used a lot of files that are accessed frequently are placed on the hard disk drive thereby increasing access time as well as increasing power consumption.

The aim of this project was to see if artificial intelligence can be utilized to optimize the access time for storage devices [1]. An artificial intelligence mechanism was created that utilized several parameters provided by the trace data to make decisions to reduce the access time for files that may be used in the future. Specifically temporal locality, process data correlation and process-process correlation were considered and an algorithm was designed to reduce access time keeping those parameters in mind.

II. ANALYSIS MECHANISM

A. SSD Structure

The solid state disk is modeled as a queue where the files are sorting according to recency. That is to say the first file to be added to the queue is the least recently used file and is at the head position. As new files are to be moved to the solid state disk they are enqueued. Each file is broken into 4 Kb chunks and each position in the queue holds a 4Kb chunk. This was done due to the following reasons:

i) The queue has a fixed capacity of 2GB that is 500 entries of 4Kb each. When a file needs to be deleted a certain number of entries can be removed from the queue without having to deal with total file size of each file.

ii) It is easy to model the movement of data when it has been broken into chunks of 4Kb, each of which occupies one index in the queue. This makes it easier to move the whole file rather than dealing with variable file sizes for different indices if each index was taken as a file of a particular size. This model greatly reduces the time the simulator takes while processing trace data.

B. *Updating SSD*

There are possible scenarios that might be encountered when the SSD needs to be updated:

i) Initialization: The first time the machine learning is used the solid state disk is completely empty and no files have been read. The simulator then starts processing the data and keeps track of the current file being used; when it is closed the file is enqueued to the SSD queue until the solid state disk cannot accommodate more files.

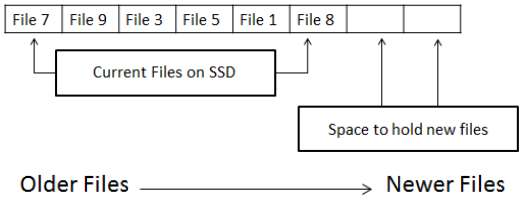

Figure 1: SSD Initialization

ii) Update: File already exists on the solid state disk and is read again. Since the file already exists but is read again it becomes the most recently used file. Once it is read and closed it is enqueued to the SSD queue since it is the most recently used file.

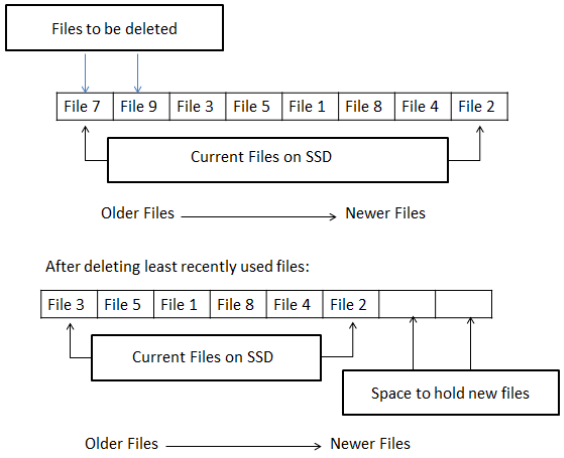

Figure 2: SSD Update and File Addition

iii) Addition: File is currently on the hard disk drive and needs to be moved to the solid state disk. Since there is no space available to add the file, a file must be deleted from the queue. Due to the Update mechanism the least recently used file is always at the head of the queue and the files are sorted according to recency. The files at the head of the queue keep getting dequeued until enough space is available to add the new file; this is done because the head holds the least recently used file, now there is enough space to add a file so the file from the hard disk drive is enqueued to the SSD queue.

C. *Time Calculation*

The time calculation mechanism depends on the location of the file. The simulator, based on the system call encountered determines whether a read to disk or write to disk was executed. For the sake of simplicity every action was classified into a read or a write and time was calculated accordingly. The time calculation was an iterative process with parameter input of File Name, R/W, bytes used. Every policy described below maintained its own SSD queue. Based on the policy, the location of file was determined and the appropriate SSD or HDD time was calculated. Due to a lack of deterministic file size information, the location of the bytes (SSD or HDD) could not have been determined with certainty. In order to estimate this, the following formula was used.

Ratio = (File Size on SSD) / (True File Size)

```
if(rand() > Ratio)
        HDD Time Calculation
else
        SSD Time Calculation
```

The following numbers were used for time calculations [2]:

SSD Read = 250 MB / s
SSD Write = 70 MB / s
HDD Read = 15 MB / s
HDD Write = 7 MB / s

To simplify time calculation the transfer time for moving a file from HDD to SSD has not been accounted for. This transfer time will be mainly invisible to the user, given the scheduling property of operating systems.

III. POLICIES

A. *Process-File Correlation*

This policy relies on a direct and predictable relation between a certain process and the files it initiates. Section VI (B) describes the simulator that maps every file to the respective process that initiates it. In the simulation of this algorithm, every time the user initiated a new process, the historical file record of the

process was transferred onto the SSD. The process of updating the SSD was standard and similar to other policies and as described in (ii) of this section. The Process-File policy had caching enabled. This essentially means that as long as the SSD had space to fit in additional files, the newly read file would be transferred onto the SSD. Once the SSD is filled up to capacity, the algorithm starts and the least recently used files are ejected to accommodate for the files corresponding to the active process. The file requirements for a given process are bound to change over a period of time. It is also physically impossible to load more that 2GB of files for a certain process. Therefore similar to maintaining the SSD queue structure as described in (i), the simulator generates a queue of files capped at 2GB. The structure instead of containing 4KB nodes as in the case of SSD contains filenames. This is done in order to compensate for the lack of file size. It must be noted that the size of a certain file might increase by a write() using another process. A detailed explanation of file size estimation can be found in Section VI (B). The policy is implemented as a function within the simulator and returns the time to execute the system call for every iteration of the trace data. The time calculation is based on the location of the file, whether on SSD or HDD, and file size. The time calculation is standard and similar to other policies and has been described in (iii) of this section.

B. Frequency-File Correlation

The parameter under consideration in this policy is the frequency of a particular file. The files with large frequencies get qualified for placement on SSD. This policy has an initial caching enabled as described in (ii). Once the SSD is filled, a threshold frequency is assigned for the files. The initial threshold frequency is set at 5 initializations over an infinite period of time. As the SSD gets updated with user activity, the frequency also increases. Since, the frequencies never get reset, there is no cap on the value the frequency of a file could attain. In order to account for this, the threshold frequency is dynamically updated. This updating mechanism is directly dependent on the recency of files as described in (ii). This dynamic update makes sure that no more than 2GB of user files has a usage frequency higher than the threshold frequency. Therefore if the frequency of a file qualifies it to be placed on the SSD, the SSD update function (i) ejects the least recently used file. The frequency of the file that is ejected is the new threshold frequency. Hence, over a period of time the older files clog the SSD and are replaced by equally old files.

C. Time- File Correlation

Time stamps were used in order to determine what files were to be placed on the solid state disk. Using the time stamp information from the trace data and the current time of the system the algorithm decided whether the file should be placed on the solid state disk or not. The files to be stored on the solid state were divided on an hourly basis i.e. the files that were required at a certain hour of the day were placed on the solid state drive before it became that time of the day. For example the files required at 10 A.M were placed on the solid-state drive at 9.50 AM. This policy was implemented within the simulator and it returned the total time needed to execute the given trace data. The solid state disk is implemented as described in section (i) and the update functionality based on the recency of the files was implemented as described in section (ii).The time calculation was based on the location of the file SSD or HDD and the file size. The time calculation is implemented as described in section (iii).

D. Continous Caching

The solid state disk was used to implement caching. The solid state disk was implemented using the queue structure described in section (i). The continuous caching mechanism used the Update SSD mechanism described in section (ii) for each file that was read. The time calculation is based on the location of the file SSD or HDD and the file size. The time calculation is implemented as described in section (iii).

E. Time-Process-Frequency Correlation

Amalgamating the one attribute dependency of the first three policies a time-process-frequency dependent policy was created. In this policy every process was mapped to a one-hour time slot determined by the time at which the process was initiated. If the same process was initiated at two different hours, duplicate entries were made for the process in the time log. Based on the time of day, and frequencies of the files of the process, probabilities were assigned to the pool of files. The probabilities were the best estimates whether the file would be needed on the SSD eventually. This policy had higher accuracy, given the fact that the user tends to run a particular process at a certain time of day rather than a set of files. Through the Process-Time correlation, a better prediction was made about

which files the process would require, given that time of the day and the user's process patterns. This policy is further described in the Bayesian Network analysis in Section V.

IV. FINDINGS

A. Training Data: 2 weeks

Policy	Total Time Taken	Savings*
SSD	3.3 hours	88.6 %
HDD	28.9 hours	0 %
Process-File	20.87 hours	27.8 %
Process-Time-File	19.91 hours	31.1 %
Process-Frequency-File	20.49 hours	29.1 %
Frequency-File	26.79 hours	7.3 %
Time-File	23.24 hours	19.6 %
Time-Process-File	18.84 hours	34.8 %
Time-Frequency-File	22.63 hours	21.7 %
Time-Process-Frequency-File	16.73 hours	42.1 %
Continuous Caching	27.39 hours	5.2 %

B. Testing Data: 1 year

Policy	Total Time Taken	Savings*
SSD	157.46 hours	90.1 %
HDD	1560.20 hours	0 %
Process-File	1165.47 hours	25.3 %
Time-File	1280.93 hours	17.9 %
Time-Process-File	1045.33 hours	33.0 %
Time-Frequency-File	1210.72 hours	22.4 %
Time-Process-Frequency-File	939.24 hours	39.8 %

*: Saving compared with regular drives (HDD)

V. DESIGN

A. Planning Algorithm

This model is based on a Plan-Space Planner [3]. The planner starts with an empty plan and adds constraints that are parameters to be used by the discriminative model. At each level of the plan space planner a parameter was added as a constraint. The following constraints were added to the planner: Time, Frequency, Process, and Cache. These were the only constraints that were added since the trace data featured information only about the Time, Process and File and from this information frequency could be inferred. However, File was not added as a constraint since the decisions being made were about the locations of the File. Relations were derived between Time-File, Process-File and Frequency-file and the decisions depended upon the Time, Process and Frequency and not directly upon the file. The Plan-Space Planner used two weeks of trace data to create this plan. However, when the algorithm is deployed this plan would use data being acquired dynamically rather than two weeks of trace data that is being used right now.

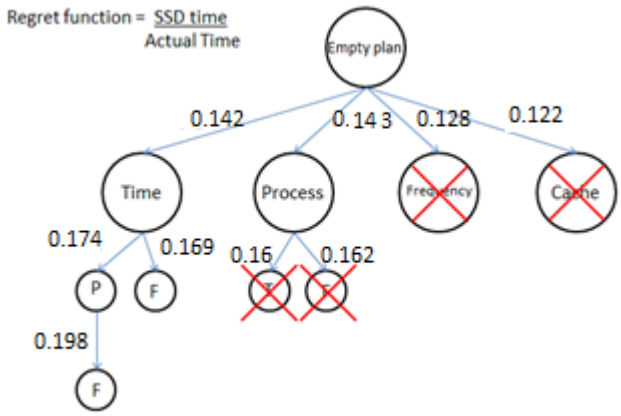

Figure 3: Plan-Space Planner

The regret function for each plan was calculated as SSD Time/ Actual Time. Therefore this value will always be between 0 and 1, not inclusive. At each level the two partial plans with the lowest regret functions were eliminated. Thus, at the first level Cache and Frequency plan spaces were eliminated. Frequency and Caching displayed very poor results this was because Caching simply placed files on the solid state disk after they were read once and in a large number of cases these files were never read again, thus having them on the solid state disk was not beneficial. Frequency performed poorly as frequency changes dynamically but older files were given preference and placed on the solid state disk. Even if a new file had the same frequency as an older file, the newer file was not added to the solid state disk thus not reducing access time. Then the partial plan spaces of Process were further constrained with Time and Frequency. Similarly, Time was constrained with Process and Frequency. Cache was disregarded as it yielded a very high actual time and did not seem suitable in the partial space of Time and Process. The regret functions were calculated again. Based on the regret function Process-Time and Process-Frequency were further eliminated since they

had the two lowest values. Process-Time yielded poor results since the files used by a process varied from based on times. For example an Update process during startup of the machine used different files than the same Update process that ran later during the day, thus the files being used by processes were being changed dynamically and had a weak dependence on time yielding poor results. Process-Frequency yielded poor results since the frequency of a process did not reflect how many files the process used and there was no dependence between the actual number of files used by a process and the frequency of the process. For example a process that uses 1 file could be called a 100 times, so based on the frequency the file would be placed on the solid state disk whereas a process that uses 300 files and has a frequency of 20 was not be placed on the solid state disk since it had the lowest frequency, thus the file that was placed on the solid state disk is accessed a 100 times but the potential access times of the 300 files 20 times has been missed.

The final plan of Time-Process-Frequency yielded the best regret ratio. The Time-Process-Frequency plan gave the best result because during a certain time period during the day the same processes were running and these processes were using the same set of files. Placing these files on the solid state disk according to the frequency they were accessed allowed the files that were accessed most frequently by a process during a certain time period to be on the solid state disk which gave quick access to a very large percentage of the files that were being accessed thus giving very good result and achieving a large time saving.

B. Discriminative Model

The model was discriminative. There were three given parameters: Time, Frequency and Process. The model was used to depict the dependence of the unobserved variable which is the location of the file on the observed variables that are time, frequency, and process. The observed variable was one of the three parameters and a prediction was made for the location of file. Based on the observed variable a direct estimation was made about the future location of the file. This model was discriminative as once the decision was made it was not possible to get information about why a specific file had been placed in that location, there was no attempt to model the underlying probability distributions [3]. The probability distribution was not modeled due to the large number of files that were being handled by the simulator. It would be physically impossible to maintain a joint probability distribution for larger than a thousand files and their correlations with time, frequency and process.

C. Bayesian Network

As mentioned in Section 2, the first four policies are simple one parameter dependent, while the Time-Process Frequency correlation incorporates 3 variables. Therefore the need for creating a Bayesian network arises. Figure 4 is an illustration of a Bayesian Network for a Process-File correlation. Given a one parameter input there exists a simple parent child relation in a tree of depth 1 and hence the probability has no significance.

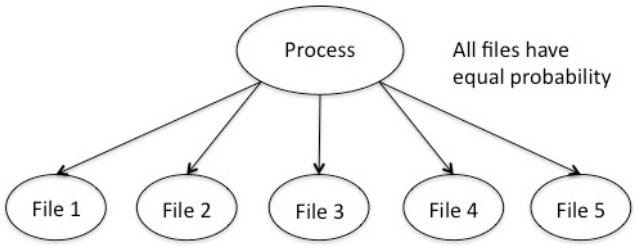

Figure 4: Process-File Tree

However, in Figure 5 the Time variable is the head of the tree, and determines the processes initiated in the one-hour span. Given a one-hour time granularity is not deterministic of the processes that might be initiated in the future, the following hour were included as well. It must be noted that the current T hour information has not been considered in the Bayesian network for T+1. This is because the files transferred for hour T will be the most recent on the SSD and will therefore be the last to get eliminated from the SSD by the Update SSD function described in Section 2. At current time of T hours and 50 minutes, the simulator generates a tree using the parameter hierarchy obtained through the Plan space planner. Currently, the simulator was hard coded to create Bayesian networks for the parameters identified using the planning algorithm. In the future, when the algorithm is deployed a generic network of variable size can be created based on the information variables selected by the Planning algorithm. This will especially be helpful when more information can be inferred from the operating system trace data. Currently the tree generated by the simulator is an amalgamation of the trees for time T+1 and T+2. Since the policy is initiated every hour, at T-1 hours, the files for T+1 get added. These files again try to get added at time T hours. Therefore technically the 1st priority of the fill-up order on the SSD will already exist. Thus every hour the second priority of the fill-up order gets added.

This might seem futile at first; however, it does not have any attached costs and ensures that a minimum two hours of future buffer is maintained on the SSD. It is also important to note that the files (leaves F1-F9 in figure) for every T node are capped at 1 GB capacity, thereby making it impossible to erase a T+1 file at time T hours [4].

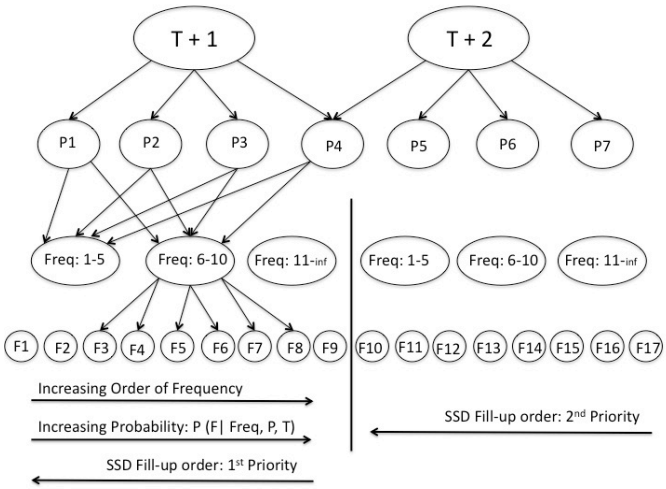

Figure 5: Time-Process-Frequency-File Tree

VI. EXPERIMENTAL SETUP

A. Historical Data

The analysis described in Section 3 incorporated trace data at the operating system level. Such user activity data is abundantly available through studies worldwide. However, for the purposes of this project the trace collection of SEER Predictive Hoarding System study at UCLA was selected [5]. The trace collection begins on June 21st 2001 and contains activity for about a year's duration. A snippet of the data is described below. The data in the study is space separated.

2588 UID 62936 PID 881 /bin/sh A 960473274.891599 open("/lib/libc.so.6", O_RDONLY, 4016683) = 3

2668 UID 62936 PID 881 /bin/sh B 960473274.891714 close(3, 4016683) = 0

a. Offset: The number describes the record offset value. This value was generated for collection purposes and is therefore ignored in the study
b. UID
c. Unix User ID
d. PID
e. ID of the processes initiating the system call. Since UNIX reuses the process IDs this value is not a unique identifier. Therefore PID is ignored in the simulations.
f. Process name generating the trace record. Since the name is unique for a process, it is used in the analysis to map the files to respective processes.
g. The flags B, A, S or G correspond to the respective time the trace was collected compared to the system call execution. This value is important from the perspective of an operating system study and has thus been ignored.
h. Unix Time Stamp. The value is dot separated. The second half corresponds to the microseconds value of the system call. Given the study analyzes a broad time to process correlation, the microseconds have not been considered.
i. The field preceding the brackets lists the system call being generated
j. The arguments passed to the system call are listed in the brackets. For example the "open" system call passes a file name.
k. The last value for most system calls is the return result value. This is 0 (success), -1 (failure), file descriptors, process IDs or the bytes accessed through the system call.
l. In certain cases an error is printed after the result. A system call entry generating errors has been ignored from the analysis.

B. Simulator

A Java based simulator was setup to parse the operating system traces. The simulator ran through each entry and stored the User ID, Process name, time of system call, system call, file name and any relevant return value. The information extracted through each activity, was stored in a two-layered structure as illustrated in Figure 6. The historical data described in the previous section was parsed and stored iteratively. The two-layered structure is as follows:

i) Process -> File:

Each file read was mapped to a certain process with the respective timestamp. A process entry had multiple files that the process initiated over a period of time. The size of the process queue, that is the quantity of files each process holds was restricted to 2GB. Hence an entire process could occupy the SSD, or it could be split up into various processes.

ii) File -> Activity:

The activity of each file was stored using a linked list mapped to a particular file. Every system call, irrespective of the process, that involved the file was mapped to it. However, in the analysis this information was rarely used. This part of the structure could be incorporated in future work.

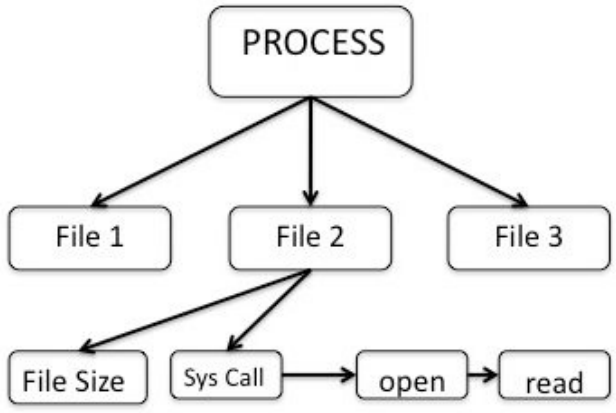

Figure 6: Simulator Information Storage

Storing the OS activity data in this structure had a significant time cost attached given there are on average 7-10 system calls per second. Hence running through a year of historical data involved several hundreds of thousands of iterations. It must be noted however, that the time cost per iteration was relatively low. This is important, given the fact that, in deploying this system to the user machine, the time cost will factor in per iteration rather than for running several iterations.

C. Drawbacks

Determining the file size was a particular challenge faced in creating the experimental setup. As is evident in part a) of this section, file size information was not collected during the SEER trace study. Therefore the simulator iteratively calculated the size of each file. When a file was initiated for the first time, the simulator kept a counter of how many bytes were being accessed and at the close() system call, the counter was stored as the file size. In subsequent activity, if the file was initiated again, a new counter was setup. If the counter reached a value larger than the current file size, the file size was updated with the counter. This calculation was a rough estimate and might be prone to error. However, the error does not impact the findings of this project heavily. This is simply because the presence of the file on the SSD (or HDD) is the quantifier to the success/ failure of the algorithm rather than size on SSD. Nevertheless, size was calculated through the most efficient predictions.

VII. Future Work

A. Online algorithm

The algorithm will proceed in a sequence of iterations. First, the algorithm will receive information about a file: timestamp, process, frequency of the file. In the second step, based on this information the algorithm will make a prediction using the decision tree mechanism described in the section below. In the third step of the algorithm feedback will be provided specifying if the file was actually used at that time or not. Given this information about whether the file was actually used or not the decision tree will be altered to improve the prediction for files in the future.

Since the learning algorithm receives feedback about the labels continually, the algorithm will be able to adapt and learn even in difficult situations. Most online learning algorithms do not receive accurate label feedback; however for our problem the algorithm will receive the correct label. The feedback about whether a file was used or not at a specific time will be provided in a set which specifies which files were used within a time period of one hour. The algorithm will receive 24 such sets during the day, ensuring that the decision tree classifier can be improved using the information about the usage of the file.

B. Decision Trees

Given the implementation of an online learning mechanism, the feedback for every particular decision will be obtained by the system. Therefore, in the Time-Process-Frequency policy that gets updated every hour, it can be known at the end of the hour whether the decision that was made was correct. With the 24 sets of feedback data per day, over a period of time the learning data will be fairly large and can thus be organized into highly informative decision trees. The file information that can be inferred from the data as mentioned previously is Time, Process and Frequency. More parameters may be introduced into this decision tree structure while running the algorithm on a more informative trace dataset. Following is an example of the decision tree structure [6].

Time:
EM – Early Morning = 4am-10am
DY – Day = 10am-4pm
EV – Evening = 4pm-10pm

NT – Night = 10pm-4am

Process:
OS – Operating system processes (Daemon)
UI – User initiated processes
BU – OS Boot up processes

Frequency:
BT – Below Threshold Frequency
T1 – Threshold Frequency + 10
T2 – T1 + 10
T3 – T2 + inf

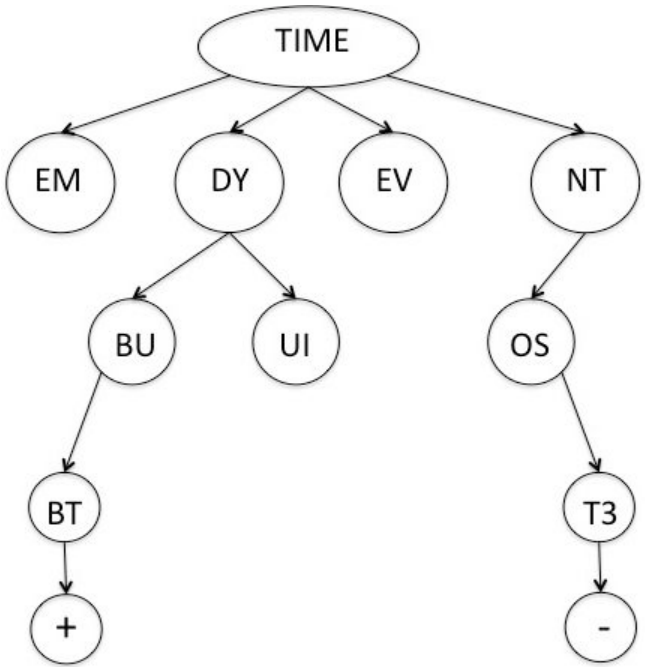

Figure 7: Decision Tree Based Implementation

A description of the Threshold Frequency is provided in Section III. B. Frequency-File correlation. The decision tree is initially created based on the output of the Plan Space planner described in Section V. A. Planning algorithms. Upon the creation of a basic structure, the decision tree is populated with feedback data received every hour. The Information gain will then be calculated for each depth to quantify the performance of the structure. A classification of the attributes in this manner will help in determining certain patterns that might be hidden in a mere process to file relation etc.

C. Cross Validation

Once the decision tree model has been generated, cross validation can be used to determine how the model will be perform in practice. K-fold cross-validation with K = 10 can be used to determine the predictive performance of the algorithm. The trace data will be split into 10 subsets, one set will picked for testing and the remaining nine sets will be used as training data for the decision tree. This will be repeated 10 times using each one of the sets as testing data exactly once ensuring all subsets have been used for validation and training. The results from these tests will then be average to give an estimation of the performance of the decision tree. Based on this performance measure the decision tree can be modified to use different parameters at different levels of the tree. If the current decision tree described in the next section yields a poor performance then either process or frequency or any other parameters based on more informative trace data could be used at the root level and different depth levels to create the decision.

REFERENCES

[1] SynergyFS: A Stackable File System Creating Synergies between Heterogeneous Storage Devices: http://www.kernel.org/doc/ols/2008/ols2008v2-pages-255-260.pdf

[2] Seagate Momentus XT: http://www.seagate.com/www/en-us/products/laptops/laptop-hdd#tTabContentSpecifications

[3] CS 440 Lecture Slides- Prof. Gerald DeJong: http://www.cs.uiuc.edu/class/fa10/cs440/

[4] Machine Learning University at Buffalo: http://www.cedar.buffalo.edu/~srihari/CSE574/

[5] SEER Predictive Hoarding System- Prof. Geoff Kuenning, UCLA: http://lasr.cs.ucla.edu/geoff/seer.html

[6] Statistical Data Mining Tutorials- Prof. Andrew Moore, Carnegie Mellon University: http://www.cs.cmu.edu/afs/cs/Web/People/awm/tutorials/