Troubleshooting Thousands of Jobs on Production Grids Using Data Mining Techniques

David A. Cieslak, Nitesh V. Chawla, and Douglas L. Thain
Department of Computer Science and Engineering, University of Notre Dame

Abstract

Large scale production computing grids introduce new challenges in debugging and troubleshooting. A user that submits a workload consisting of tens of thousands of jobs to a grid of thousands of processors has a good chance of receiving thousands of error messages as a result. How can one begin to reason about such problems? We propose that data mining techniques can be employed to classify failures according to the properties of the jobs and machines involved. We demonstrate this technique through several case studies on real workloads consisting of tens of thousands of jobs. We apply the same techniques to a year's worth of data on a 3000 CPU production grid and use it to gain a high level understanding of the system behavior.

1 Introduction

Large scale production computing grids introduce new challenges in debugging and troubleshooting. Consider the user that submits ten thousands jobs to a computational grid, only to discover that half of them have failed. The failures might be due to a bug in the program on certain inputs, an incompatibility with certain operating systems, a lack of resources to run the job, or just bad luck in the form of a widespread power failure. How can a user or administrator of a large computing system diagnose such problems? It does no good to examine one or even a handful of jobs in detail, because they may not represent the most common or the most significant failure mode.

We propose that data mining techniques can be applied to attack this problem. Computing grids of various kinds already publish a large amount of structured information about resources, users, and jobs. Given access to this data, classification techniques can be used to infer what properties of machines, jobs, and the execution environment are correlated with success or failure. For example, it should be possible for the user with 10,000 jobs to quickly discover that 90 percent of all failures occurred on Linux machines running kernel version 2.6.2 with less than 1 GB of RAM, and the remainder were randomly distributed.

Of course, this kind of analysis does not immediately diagnose the root cause of the problem, but it would be a significant help in allowing the user to focus their attention on the situation or resource of greatest interest. In this case, the information would allow the user to quickly identify the correct environment in which to debug their program in detail. Or, the user might simply direct the grid computing system to avoid machines matching those properties.

In a previous short paper [4], we introduced the concept of troubleshooting large production runs using data mining techniques. We demonstrated the feasibility of this approach by executing large synthetic workloads with induced failure conditions dependent upon the properties of execution machines. Our techniques were able to infer the properties of machines correlated with failures, which were closely related to the root cause.

In this paper, we apply our techniques to real workloads in which trouble is suspected, but the root causes are unknown and not easily diagnosed by the user. We consider a number of workloads of tens of thousands of jobs each submitted to a 500-CPU system at the University of Notre Dame, showing examples of problems diagnosed by examining machine properties, job properties, and the relationship between the two. We then apply these techniques to data collected over the course of a year on 300,000 jobs running on over 3000 machines on a large campus grid at the University of Wisconsin. Throughout, we discuss the strengths and limitations of this approach.

2 Troubleshooting Technique

We have implemented a prototype troubleshooting tool that diagnoses workloads submitted to Condor [22] based grids. Figure 1 gives an example of the input data to our troubleshooting tool. Every job submitted to Condor is represented by a ClassAd [18] that lists the critical job properties such as the owner, executable, and constraints upon machines where it is willing to run. When the job is complete, the exit code, resource consumption, and other statistics are added to the ClassAd. Every machine participating in Condor is also represented by a ClassAd that lists the available

Job ClassAd		
JobId	=	29269
Owner	=	”dthain”
VirtOrg	=	”NWICG”
Cmd	=	”mysim.exe”
Args	=	”-f -s 5”
ImageSize	=	82400
ExitCode	=	1
ExitBySignal	=	FALSE
Requirements	=	(Arch==”LINUX”)
Rank	=	(MachineGroup==”elements”)
Machine ClassAd		
Name	=	”aluminum.nd.edu”
OpSys	=	”LINUX”
Arch	=	”INTEL”
IpAddr	=	”111.222.333.444”
TotalDisk	=	3225112
TotalMemory	=	1010
Mips	=	3020
LoadAvg	=	1.5
KeyboardIdle	=	272
MachineGroup	=	”elements”
Rank	=	(UserGroup==”physics”)
Start	=	(KeyboardIdle>300)

User Log

(29269) 05/02 09:35:53 Submitted 111.222.333.444
(29269) 05/02 09:44:13 Executing 111.222.333.555
(29269) 05/03 17:57:53 Evicted.
(29269) 05/03 17:59:01 Executing 111.222.333.666
(29269) 05/04 04:35:02 Exited normally, status 0

Figure 1. Sample Data for Troubleshooting

physical resources, the logical configuration of the machine, and constraints upon jobs that it is willing to accept. The ClassAd language is schema-free, so the precise set of attributes advertised by jobs and machines varies from installation to installation, depending on what policies the local administrator constructs. However, within one Condor installation, there is a high degree of schema consistency.

A *user log file* gives the sequence of events in the lifetime of a workload. The log file states when each job is submitted, when and where it begins executing, and when it completes. It also records the following undesirable events that are of great interest for troubleshooting. When a job completes, it indicates whether it exited *normally* (i.e. completed `main`) with an integer exit code, or *abnormally* (i.e. crashed) with a given signal number. A job can be *evicted* from a machine based on local policy, typically a preemption in order to serve a higher priority user. A job can suffer an *exception* which indicates a machine crash, a network failure, or a shortage of system resources. A job can be *aborted* by the user if it is no longer needed. In this work, we define *failure* as any non-zero completion, abnormal completion, eviction, or exception.

Between these three data sources, we can observe what jobs succeed, which fail and what the properties of the jobs and machines involved are. Each of these data sources is

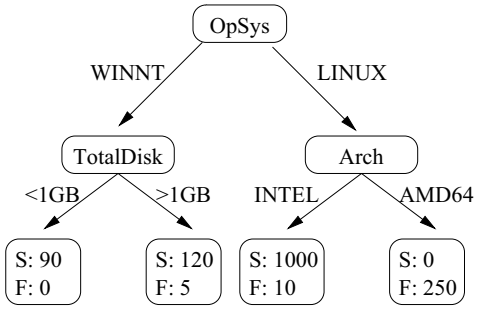

Figure 2. Sample Decision Tree

available to end users without any assistance from the administrator, so this technique can be widely applied. Users may diagnose workloads that are incomplete. Both the job and machine data are collected at the time tool is run; this imposes some limitations we discuss below.

A challenge in this domain is that the failure and successes are not constant — that is, the typical production grid observes a varying degree of failures and successes over time, which is largely driven by the addition/deletion of users and applications. This also imposes the issue of high class imbalance in the data [2]. An analysis of the workloads below shows that the failures and success are varying over time significantly. Thus, the primary challenges are to be able to counter the issues of class imbalance, data distribution, and model interpretability.

Standard decision tree algorithms can be very sensitive to imbalance in class distributions [9], thus limiting their use in our work. To that end, we implemented a decision tree algorithm that utilizes the Hellinger distance as the objective function [5]. Hellinger distance is invariant to the class skew; therefore, it is ideal for problems in which the class ratios are unknown and potentially dynamic. This algorithm is highly effective under class imbalance, trains quickly compared to other imbalance solutions such as sampling, and generates readable model.

Our prototype tool accepts the three data sources as input, and then constructs a *decision tree* for both job and machine properties, the Hellinger Distance Decision Tree (HDDT) [5] algorithm. A sample tree is shown in Figure 2 The leaves of the tree represent disjoint subsets of jobs, a certain number succeeding (S) and a certain number failing (F). The internal nodes of the tree represent job or machine properties, and the edges indicate constraints upon the values of those properties. For example, the rightmost leaf indicates the category of jobs that ran on machines where `OpSys=="LINUX"` and `Arch=="AMD64"`. 250 jobs in that category failed, and none succeeded.

A decision tree from a real workload is much larger, and may have hundreds of leaves. Such a data structure can be difficult for even the expert to parse. To accommodate the

Static Properties (most relevant to success/failure)		
OpSys	=	"LINUX"
Arch	=	"INTEL"
Subnet	=	"111.222.333"
TotalDisk	=	3225112
TotalMemory	=	1010
KFlops	=	842536
Mips	=	3020
JavaVersion	=	"1.6.0_05"
CondorVersion	=	"7.0.2"
KernelVersion	=	"2.6.18-53.1.13.el5"
Dynamic Properties (only add noise to the result)		
LoadAvg	=	1.5
CurrentTime	=	1206568628
KeyboardIdle	=	272
Disk	=	3225112
Memory	=	1010
VirtualMemory	=	2053
Artificial Labels (can hide root causes)		
Name	=	"aluminum.nd.edu"
IpAddr	=	"111.222.333.444"
MachineGroup	=	"elements"
IsDedicated	=	FALSE
IsComputeCluster	=	TRUE
ForDan	=	FALSE

Figure 3. Categories of Machine Labels

non-expert user, our tool simply emits the three most significant nodes of the decision tree as ClassAd expressions. In our experience so far, we have found this to be effective.

In our initial attempts at applying this technique, the created decision tree were enormous, very difficult to read, and contained a number of splits on what appeared to be irrelevant data. The problem arose becuase we collected machine data by querying Condor after the workload completed, in some cases weeks or months later. Figure 3 gives examples of the kinds of data present in the machine ClassAds. *Static properties* are labels created by Condor itself according to the fixed local hardware or operating system, which is not likely to change over the course of weeks or months. *Dynamic properties* are observations of constantly changing values, such as load average or available memory. *Artificial labels* are properties assigned by administrators in order to simplify policy enforcement or reporting.

In general, it is necessary to remove known dynamic labels from the algorithm, because they only add noise to the output. As we will show below, it is meaningful to classify both static and artificial labels, but each produces a different character of results. Our prototype tool allows for both forms of analysis.

3 Single User Workloads

How do we evaluate the quality of a troubleshooting technique? Quantitative performance evaluation does not apply: the techniques presented here run in seconds on workloads of tens of thousands of jobs, and in minutes on the whole-grid data presented later. We may judge a technique to be useful if it correctly diagnoses a problem that is known but not understood, or reveals a problem that was entirely unknown. Of course, no single troubleshooting technique is universally applicable. To evaluate this technique, we present several case studies of real workloads, and use them to describe the limits of the technique.

We begin by considering single-user workloads executed on our local campus grid of 500 CPUs at the University of Notre Dame. This grid is highly heterogeneous, consisting of multiple dedicated research clusters and cycle-scavenged desktop and classroom machines. Machine performance and capacity varies widely: CPUs range from 500-6000 MIPS, disks range from 400MB to 500GB, and RAM ranges from 243MB to 6GB. In such an environment, diagnosis of incompatibilities is critical.

Our first case study is set of 60,000 jobs that perform a biometric image processing workload. The jobs are generated by a high level abstraction designed for non expert users. The abstraction divides a very large problem into a small number of very similar jobs, so any problems are likely to be due to variations in available machines.

The user in question presented with the complaint that the workload had a long tail. A number of jobs at the end of the workload took far too long to complete, having been evicted many times from multiple machines.

Figure 4 shows the output of the troubleshooting tool. When run on artifical labels, it produced a decision tree too large to present in full here. However, the most significant rule immediately pinpointed the problem: these jobs were always evicted from machines where `MachineGroup=="ccl"`, a label indicating a cluster of seven homogeneous machines.

The results on static labels give a better sense of the root cause. These machines are distinguished from others in the pool by three properties: `TotalDisk <= 125GB` and `JavaMFlops <= 3.1` and `TotalVirtualMemory <= 1GB`. Based on our knowledge of the workload and machines, we know that the disk space is sufficient and the Java performance is irrelevant. However, the total virtual memory seems unusually low: most machines are configured with at least several GB of virtual memory. We hypothesized that the job was crashing due to exhausting virtual memory, and confirmed this by manual testing. The cluster was misconfigured when it was installed, but no-one noticed until these particular jobs ran.

What may we observe from this exercise? First, the artifical labels allowed for a compact, correct statement, but did not suggest the root cause of the problem. The static labels are more suggestive, less compact, and require the reader to apply some domain knowledge to get to the root cause. But,

Feature mapping	Tree properties	Most Significant Rule	Machines Indicated
Artificial Labels	height: 13 nodes: 301 leaves 166	(MachineGroup == ccl) → (completed:0 evicted:305)	ccl00 – ccl07
Static Labels	height: 19 nodes: 301 leaves 153	(TotalDisk $\leq$ 125982176) && (JavaMFlops $\leq$ 3.152064) && **(TotalVirtualMemory $\leq$ 1043856)** → (completed:0 evicted:305)	ccl00 – ccl07 **cclbuild00 – 03**

Figure 4. Decision Tree Results on a Biometric Workload

Feature mapping	Tree properties	Most Significant Rule	Administrator Conclusion
Machine Labels	height: 10 nodes: 34 leaves: 18	(TotalDisk $\leq$ 36669824) && (TotalVirtualMemory $\leq$ 4192800) → (completed: 8 ShadowException: 737)	The code generated very large output files, exceeding the OS imposed 2GB limit.
Job Labels	height: 2 nodes: 34 leaves: 31	**(Parameter1 = 4)** → (completed: 0 ShadowException: 393)	Setting Parameter1 to 4 makes linear separation for SVM training intractible.

Figure 5. Decision Tree Results on a Data Mining Workload

both forms create true statements that can be acted upon. In both cases, the expression emitted by the troubleshooter can be placed into the user's submission file verbatim in order to avoid machines with those properties, without even bothering to understand the root cause. In the case of the static properties, the requirement even matches several additional machines on which the job is likely to crash, but simply did not happen to appear in the original workload.

Not all failures can be blamed on properties of machines. In many cases, a particular job may have some property that prevents it from executing successfully on **any** machine. Can this technique diagnose problems with jobs?

Another user ran a workload of several hundred jobs, performing a study of various data mining techniques on several datasets. Much like the previous example, a large portion of the workload finished quickly, but a few stragglers remained. This problem was debugged in two stages.

Figure 5 shows the output of the debugging tool. Initial analysis on the machine properties was not fruitful. The most significant branch indicated `TotalDisk` and `TotalVirtualMemory` below critical values as associated with failure. Although that analysis was true, a lack of resources was not the root cause as it was above.

Analysis on job properties was much more fruitful. Fortunately, this user made a habit of putting logical information about the job into the ClassAd for bookkeeping purposes. First, our tool indicated that all the failures were occuring to jobs using the dataset `oil`. A manual test of jobs on this dataset revealed that, because of the large dataset size, the algorithm was producing an output debug stream that exceeded 2GB, the maximum file size for processes using the 32-bit interface. The user disabled the debug output and re-ran the workload.

This improved the workload, but still stragglers remained. Again, the tool was applied, and this time the result was much clearer. Jobs on multiple datasets with `Parameter1 == 4` were always evicted and never completed. In this particular case,`Parameter1` controlled the complexity of the algorithm; setting it to `4` yielded a runtime so high that it could not complete within the 8-12 hour window available on desktop machines. By redirecting the jobs to dedicated machines, they eventually completed.

This exercise offers the following lessons. First, that troubleshooting via data mining is not a silver bullet. In the case of the 2GB file problem, the association with the `oil` dataset narrowed the search space for the problem, but did not point directly to the root cause. Second, manual annotation of the job properties can be of significant value. By placing runtime parameters directly into the ClassAd (instead of just in the `Arguments` property), the troubleshooter was able to isolate those associated with failure.

4 Multi User Workloads

So far, we have examined several examples of debugging via data mining. In each case, a user with a large coherent workload suspects a problem, and explicitly invokes the debugging tool to investigate. Can we also use the same techniques to study a large production computing system with many different users over a long period of time?

To answer this question, we applied the same techniques to data collected from a large production grid of the course of one year. The Grid Lab of Wisconsin (GLOW) is a campus-scale grid of about four thousand machines. The grid is physically partitioned across multiple campus departments, each running Condor, all "flocked" together for mutual load sharing. GLOW is also accessible from the Open Science Grid via a Globus GRAM [6] interface, which can submit jobs to any of the Condor pools in GLOW. A remote user would typically use an agent such as Condor-

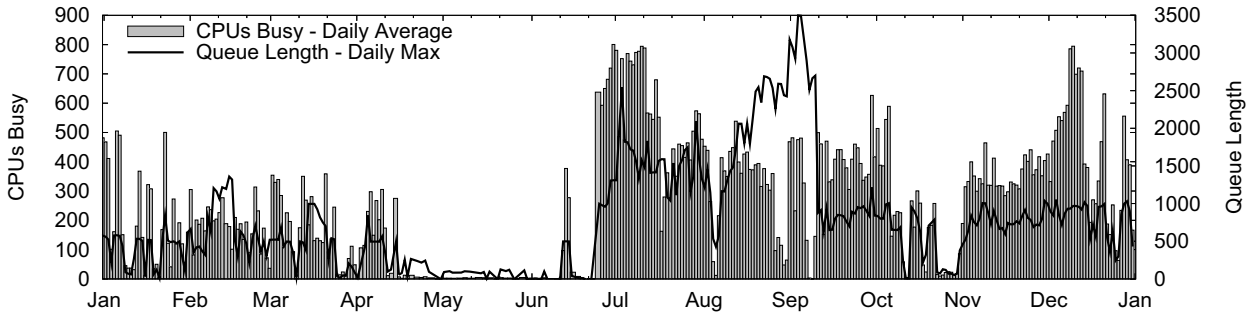

Figure 6. Timeline of CPU Utilization in the GLOW

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Jobs Submitted	14382	18469	8719	7872	1529	9234	68236	78474	25528	15595	12828	36372
Jobs Complete	13980	14873	8693	7679	1527	8230	28404	19937	23314	16127	11869	35420
Jobs Failed	365	1736	23	277	0	1	34325	56707	774	161	51	286
Jobs Rolled Back	13671	5530	8196	2366	821	1822	16827	83729	65634	88914	104606	170560
Jobs Aborted	37	1367	382	35	0	8	5050	1120	2768	79	202	523
Goodput (CPU-days)	5268	3329	4139	2196	36	3575	14391	8034	8523	5034	8617	6374
Badput (CPU-days)	523	1993	732	207	5	152	2557	2251	1524	1288	1061	3724

Figure 7. Number of Events per Month in GLOW

Virt Org	Submitted	Completed	Failed	Aborted	Exception	Evicted
cmsprod	122422	102623	16126	5698	14043	105786
cdf	119185	64203	56635	902	13992	338252
samgrid	34132	17878	11449	4848	4808	11563
nanohub	2012	1225	770	23	1596	5889
engage	1998	1563	392	52	953	7976
uscms01	1672	893	734	48	12	2962
mis	252	210	50	0	0	524

Figure 8. Number of Events per Virtual Organization in GLOW

G [11] to manage job submissions into GLOW.

GLOW is already instrumented as follows. Each individual job that arrives via the Globus GRAM interface is submitted to Condor. Each job produces a user log file that records the events in the lifetime of the job. Upon completion of the job, GLOW automatically archives the user log file and the final ClassAd of the job. In this section, we present results from all the data recorded archived 2007. Note that this data includes information about all jobs submitted to GLOW from the Open Science Grid, but it does not include jobs submitted from submitters inside GLOW itself. In addition, the machine properties were not recorded at the time of execution, so we made a single observation of machine data in April 2008.

This data is different from ordinary Condor pools in two respects. First, the "users" of the system are not recorded as individuals, but as virtual organizations. Second, the machines in the system had about thirty additional labels added in order to enforce local policy.

Figure 6 gives an overview of the load placed on this system over time. The filled boxes indicate the average daily CPU consumption of grid jobs. The dark line indicates the maximum queue length on each day. (Again, the actual utilization of the resources was higher; our data only reflects load placed on the system by remote users.) It can be seen that both load and utilization are highly bursty. Small workloads of hundreds of jobs are submitted on a weekly or monthly basis from January to May. From May to July, the system is very quiet, perhaps due to the academic calendar. From July through January, batches of thousands of jobs arrive and utilization is much higher.

Figure 7 gives more detail about the success and failure of jobs on a monthly basis. *Jobs Submitted* is the number of jobs arriving through the GRAM interface. *Jobs Completed* is the number of jobs running to completion and indicating success with an exit code of zero. *Jobs Failed* is the number of jobs running to completion, but exiting with a signal or a non-zero exit code. *Jobs Rolled Back* is the number of times in which a job is removed from an execution machine, either because it is removed by the local resource manager, preempted by the global scheduler, or fails initialize properly on that machine. In each of these cases, the job

Feature Mapping	Top Split	Most Significant
Artifical Labels	**(FileSystemDomain = hep.wisc.edu)** → (C:104656 SE:22759 E:76056)	(FileSystemDomain = hep.wisc.edu) && (TotalVirtualMemory > 10481808) && (Mips ≤ 3455) → (C:513 SE:11822 E:670)
Static Labels	(Mips > 2925) → (C:167094 SE:27732 E:152889)	(Mips > 2925) && (TotalVirtualMemory > 10481656) && (KFlops ≤ 857509) → (C:513 SE:11822 E: 670)

Figure 9. Decision Tree Analysis for All of GLOW.

Feature Mapping	Jan.	Feb.	Mar.	Apr.	May	June	July	Aug.	Sep.	Oct.	Nov.	Dec
Artificial Labels	0.212	0.312	0.558	0.407	0.407	0.402	0.717	0.526	0.772	0.531	0.627	0.551
Static Labels	0.223	0.266	0.268	0.303	0.153	0.144	0.346	0.425	0.358	0.300	0.344	0.347

Feature Mapping	cdf	cmsprod	engage	glow	ligo	mis	nanohub	usatlas1	usmcs01
Artifical Labels	0.510	0.621	0.723	0.204	0.141	0.208	0.321	0.190	0.412
Static Labels	0.387	0.395	0.258	0.063	0.198	0.154	0.285	0.053	0.182

Figure 10. Comparison of Month and User Slices to Global Data via Rand Index

is placed back in the queue and will have an opportunity to run elsewhere. A job could be rolled back many times in its lifetime. *Jobs Aborted* indicates the number of jobs explicitly removed by the submitting user. *Goodput* is the total amount of CPU time consumed that results in a successfully completed job. *Badput* is the total amount of CPU time that is consumed with any other result.

This table reveals more about the workload. First, it is very clear that the sytem goes through very large cycles of success and failure. In January, over 14,000 jobs are submitted, and most complete successfully with less than 10 percent badput observed. In February, failures and badput significantly increase, but then tail off until the summer. In July, there is an enormous increase: 68,236 jobs are submitted, 34,000 fail, and 5,000 are aborted. This trend continues in August, but failures drop significantly in September. This suggest that the initial attempts of users to harness the grid do not go well, but after repeated attempts, the proper formula is found, and a workload proceeds smoothly. For our purposes, this data confirms earlier observations of large numbers of failures in production grids.

Now, suppose that we assume the role of an administrator overseeing this large computing system. If we apply our data mining techniques, can we learn more about these failures, and use that knowledge to improve the system? This problem is more potentially challenging because multiple different users and workloads are all mixed together, and may have varying rates of failure represented with different weights according to the number of events.

That said, does it make sense to apply our classification algorithm to the entire data set? The analysis of the entire year completes in several minutes using non-optimized code running on a conventional laptop, so the technique is quite feasible for perform on a regular basis. The results are shown in Figure 9. Using artificial labels, we observe that

Label	**Info Gain**
FileSystemDomain	0.5599
CkptServer	0.4272
UidDomain	0.3329
Mips	0.3132
Arch	0.3130
HasAFS_Atlas	0.3079
TotalVirtualMemory	0.2893
IsGeneralPurposeVM	0.2856
IsGeneralPurposeSlot	0.2836
TotalMemory	0.2757

Figure 11. Most Significant Machine Labels

failure is highly correlated with `FilesystemDomain == "hep.wisc.edu"`. This property indicates the set of filesystems available from a machine. A job submitted from one filesystem domain will not have access to the data it needs, unless the user manually specifies needed data. This is a common usability problem across many Condor installations, and appears to be a significant operational problem in GLOW. Using static labels, we can observe that failure is also correlated with fast machines (`Mips > 2925`). This is a superset of the "hep.wisc.edu" machines, so the observation is correct, but not very helpful in diagnosis.

What happens if we perform the same analysis on different subsets? To answer this, we partitioned the data by month and by VO and generated decision trees for each. Rather than showing all those results, we compared each tree against the global tree using an Adjusted Rand Index, a measure of indicating the coverage similarity on a scale from zero to one [15]. As can be seen in figure 10, there is considerable variance across the dataset. In July and September and with the Engage VO, the results are quite similar. Other slices results in different top-level diagnoses.

To present these results compactly, we show the top ten properties that are most significant with respect to fail-

ure by computing information gain within the decision tree. The top five properties are all defined by the standard Condor configuration. The top three are administrative properties that describe the available filesystems, the nearest checkpoint server, and the user database. To a certain extent, all are correlated with administrative clusterings. It is somewhat surprising that `Mips` and `Arch` have equal significance. One would expect that `Arch` would have higher significance, because it describes the compatibility of programs with processors. Our hypothesis is that these properties are accidentally correlated with the more significant administrative properies, as seen in Figure 9. `HasAFS_Atlas`, `IsGeneralPurposeVM`, and `IsGeneralPurposeSlot` are artifical labels significant to the local system. This suggests that there is much to gain be defining custom properties within Condor.

Can we establish what workload or user is experiencing this problem most heavily? To answer this, we adjust the tool to alernate between job and machine properties in order to successively refine the source of failures. First, we select the subset of machines where `FilesystemDomain == "hep.wisc.edu"`, then select the set of jobs that failed on those machines, and then analyze the properties of those jobs. Despite the VO problem mentioned earlier, the most significant property of those jobs is `X509UserProxySubject`, which identifies the actual person owning the GSI [10] credentials for the job. Summing up the number of jobs, we observe:

X509UserProxySubject	**Failing Jobs**
/DC=gov/DC=fnal/O=Fermilab/CN=X	10998
/DC=org/DC=doegrids/OU=People/CN=Y	669
/DC=org/DC=doegrids/OU=People/CN=Z	145
/DC=org/DC=doegrids/OU=People/CN=W	10

To sum up, this technique, when applied to a year's worth of data from a production grid, immediately diagnosed the label most associated with failure, which also turned out to be the root cause. With guidance from the operator, the tool also identified the end user directing the workload that (depending on your perspective) suffered the most failures, or caused the most trouble.

5 Related Work

Previous authors have observed the very high rates of user-visible failures in grid computing systems. Grid3 [13] (a predecessor of the Open Science Grid) observed a failure rate of thirty percent for some categories of jobs, often due to filled disks. Iosup [16] observe that availability can have a significant impact upon system throughput. In our work, this is addressed in the sense that evictions (transition to non-availability) are a kind of failures that can be systematically identified and avoided. Schopf [21] observed from informal discussions with end users that troubleshooting is a significant obstacle to usability.

Others have performed systematic techniques for troubleshooting grids. Gunter et al. [14] describe a system for collecting large amounts of data from a grid and a technique for observing outliers in continuous valued properties. Palatin et al [17] discover misconfigured machines by correlating performance with labelled features, and selecting outliers. These have the common working assumption that performance outliers are likely to represent bugs, an approach suggested by Engler[8].

We have generally approached this problem as a matter of *diagnosing* past failures. Others have suggested *predicting* future failures. Duan et al [7] have sketched a technique for predicting future faults. Fu and Xu [12] demonstrate a technique for predicting physical failures in clusters base on time and space correlation.

Another way of troubleshooting complex systems is to study *structure* rather than *labels*. For example, DeWiz [20] examines message passing patterns between components of a distributed system to infer causal relationships. A similar idea is black box debugging [1, 19], which identifies performance bottlenecks. Pinpoint [3] tags data flowing through complex systems and applies data mining techniques to suggest individual components that are failing.

6 Conclusion

In this work, we have made the following observations:

- Production computing grids do exhibit a large number failures. From firsthand experience, we believe this is due to iteratively submitting workloads and diagnosing failures until an acceptable performance is achieved. Troubleshooting via data mining can reduce the time to successful execution.
- By constructing a decision tree and selecting the most significant decision points, we may easily draw correlations that are strongly suggestive of root causes. Both physical and logical labels are suitable for this purpose, each leading to a different perspective on the results.
- The same technique can be applied to large multi-user workloads, but the troubleshooter must have additional tools to subset and explore the data in order to draw meaningful conclusions.

And, we offer the following recommendations:

- Other sites on the Open Science Grid should adopt the logging behavior currently performed on GLOW. Although there are logs at other layers of the system, they do not contain the detail about intermediate failures stored in the user job logs.
- User log files should record the complete ClassAd of releavant machines at the time each event occurs. As

we have observed, machine properties can change between the time of execution and the time of observation, thus limiting the set of properties available for troubleshooting.

- Both job submitters and machine owners are advised to include logical information directly in classads, even if it has no immediate functional purpose. As shown in Figure 4 and 5, the non-functional labels `MachineGroup` and `Parameter1` yielded succinct, intuitive results.

This work was supported in part by National Science Foundation grant CNS-0720813. We are grateful to Dan Bradley at the University of Wisconsin; Preston Smith at Purdue University; Ryan Lichtenwalter, Christopher Moretti, Tanya Peters, and Karen Hollingsworth at the University of Notre Dame for participating in this study.

References

[1] M. Aguilera, J. Mogul, J. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging of black-box distributed systems. In *ACM Symposium on Operating Systems Principles*, October 2003.

[2] N. V. Chawla, N. Japkowicz, and A. Kolcz. Editorial: Learning from Imbalanced Datasets. *SIGKDD Explorations*, 6(1):1–6, 2004.

[3] M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox. Pinpoint: Problem determination in large, dynamic, internet services. In *International Conference on Dependable Systems and Networks*, 2002.

[4] D. Cieslak, D. Thain, and N. Chawla. Short paper: Data mining-based fault prediction and detection on the grid. In *IEEE High Performance Distributed Computing*, 2006.

[5] D. A. Cieslak and N. V. Chawla. Learning Decision Trees for Unbalanced Data. In *European Conference on Machine Learning*, 2008.

[6] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. Resource management architecture for metacomputing systems. In *IPPS/SPDP Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1998.

[7] R. Duan, R. Prodan, and T. Fahringer. Short paper: Data mining-based fault prediction and detection on the grid. In *IEEE High Performance Distributed Computing*, 2006.

[8] D. Engler, D. Chen, S. Hallem, A. Chaou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in system code. In *ACM Symposium on Operating Systems Principles*, October 2001.

[9] P. A. Flach. The Geometry of ROC Space: Understanding Machine Learning Metrics through ROC Isometrics. In *ICML*, pages 194–201, 2003.

[10] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *ACM Conference on Computer and Communications Security Conference*, 1998.

[11] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *IEEE High Performance Distributed Computing*, pages 7–9, San Francisco, California, August 2001.

[12] S. Fu and C.-Z. Xu. Exploring event correlation for failure prediction in coalitions of clusters. In *Supercomputing*, 2007.

[13] R. Gardner and et al. The Grid2003 production grid: Principles and practice. In *IEEE High Performance Distributed Computing*, 2004.

[14] D. Gunter, B. L. Tierney, A. Brown, M. Swany, J. Bresnahan, and J. M. Schopf. Log summarization and anomaly detection for troubleshooting distributed systems. In *IEEE Grid Computing*, 2007.

[15] L. Hubert and P. Arabie. Comparing partitions. *Journal of Classification*, pages 193–218, 1985.

[16] A. Iosup, M. Jan, O. Sonmez, and D. Epema. On the dynamic resource availability in grids. In *IEEE Grid Computing*, 2007.

[17] N. Palatin, A. Leizarowitz, A. Schuster, and R. Wolff. Mining for misconfigured machines in grid systems. In *International Conference on Knowledge Discovery and Data Mining*, 2006.

[18] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *IEEE Symposium on High Performance Distributed Computing*, July 1998.

[19] P. Reynolds, J. Wiener, J. Mogel, M. Aguilera, and A. Vahdat. WAP5: black box performance debugging for wide area systems. In *Proceedings of the WWW Conference*, 2006.

[20] C. Schaubschlager, D. Kranzlmuller, and J. Volkert. Event-based program analysis with de-wiz. In *Workshop on Automated and Algorithmic Debugging*, pages 237–246, Septmber 2003.

[21] J. M. Schopf and S. J. Newhouse. Grid user requirements 2004: A perspective from the trenches. *Cluster Computing*, 10(3), September 2007.

[22] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley, 2003.