Real Time Analysis and Event Prediction Engine

Joshi Fullop, Ana Gainaru, Joel Plutchak
Blue Waters project
National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign
{jfullop, againaru, plutchak}@ncsa.illinois.edu

***Abstract*—With the cost of operating extreme scale supercomputers like Blue Waters growing to what they are now and will continue to be in the future, the act of predicting failures and reacting accordingly to prevent the loss of effective compute hours and their associated power and cooling costs is quickly becoming actuarially feasible. Forecasting the systems "weather" and predicting an exact event by location, time and probability are two completely different propositions. The first provides little more than a level of awareness under which to operate the machine, and the second provides specific, localized information on which actions can be taken in advance of occurrence. We have endeavored to create a system that does the latter for Blue Waters.**

Herein we describe our utilization of specific enabling technologies and our techniques to turn an overwhelming, theoretical task into an engineered possibility. Given NCSAs history of building machines with the full spectrum of hardware, architectures and software components, we have abandoned the traditional regular expression engines for an intelligent, self-modifying template system that we created in-house, named the Hierarchical Event Log Organizer (HELO). It allows us to tag each event occurrence (log message) with an integer key identifying it as a unique type of event. This makes the process of mining the logs for event occurrences a much more manageable task since it no longer requires string matching and bulk data traversal. This benefit enables us to do an all-to-all event correlation analysis and store found correlations in a table that tracks the relationship in terms of average time between events, the standard deviation, and the confidence. With this set of correlations, we construct a directed graph to visually describe the inter-relationships between events across all subsystems. For example, an event that is reported in networking logs may precede an error in the parallel file system.

Furthermore, we use this graph to work backwards from events of interest and watch for the occurrence of any events that precede it. From the occurrence of any preceding event in any one of the found paths, we can traverse the graph forward, summing average time between events and multiplying their confidence values to deliver a probability that the event of interest will occur at a given time. The standard deviation values further allow us to provide a statistical window of likelihood. This system is also self-updating and functions in real time.

***Keywords*-event analysis; fault tolerance; fault prediction; event correlation;**

I. Introduction

In the world of extreme scale supercomputing downtime costs significant amounts of operations capital as well as reputation of the site and vendors. Power and cooling costs are essentially a complete waste not only for the downtime period, but also for any un-recoverable jobs that were taking place on the system at the time of failure. One of the largest constraints of massive scale clusters is the mean time between failures for any component in the system vs the runtime of the job. As cluster size increases, so does the probability of a single component failure within a time frame. Check-pointing jobs is an area of significant research at the moment. But even when implemented in a satisfactory manner, there is still a considerable loss of compute time back to the last check point, not to mention the time spent creating checkpoints and not on application computation. A predictive system could cause or at least give grounds to cause a checkpoint when it appears that a fault is likely. This could save a great deal of compute cycles and associated energy costs spent on the act of blindly check-pointing in fear of a future failure.

II. Background

Failures are the focus of analysis from many different perspectives. Systems administrators want to know what happened that caused systems to stop working and what to do to fix it. Site managers want to know to account for the downtime. And vendors want to understand why the failure occurred and what the conditions were so that they can improve their product. Users are generally happy to know that the system failed and there is not a problem with their code, but would like to know when they will be able to run jobs again. In trying to understand and describe a failure and how it is represented in the data that is generated by a computer (or other) system, we see that it is inherently an event.

There are two general types of data that can be monitored, events and metrics. Events are specific things that occur at a certain time and at a certain location or within a certain domain. Metrics are measurable values that describe the state of that which is being monitored. Metrics are often recorded and trended in order to determine the general health of a system. Metrics are also oftentimes the basis on which events are generated. The classic example is a simple threshold event where a metrics value exceeds some set point. Some further disambiguation as this point is necessary to distinguish between an *event* and an *occurrence* of an instance of that event. We will define, for the purposes of this paper, an *event* to be the description of what has happened. The *occurrence* is the instance when and where the event takes place. To illustrate this, well take the example of Oktoberfest. It is an event that occurs every year in late September in Munich, Germany. So in this example Oktoberfest is the event, and Oktoberfest'12 is a specific occurrence since it has a time and location. In practice, syslog provides distinct values for the time and location. However, the 'what' is left open for description by the message string. The message body generally is not distinct as it usually passes other relevant information in the body of the message. For example, "*CPU temperature critical: 91.3C*" might be a syslog message generated when a processor temperature exceeds 85C. The event of the temperature exceeding 85C can show up with a syslog message of "*CPU temperature critical: 90.5C*" as well. So we have shown two different manifestations of the same underlying event. So the trick is deriving the event given the manifestations in string form.

Some highly integrated systems offer integer based IDs in their event reporting systems. This is a most wonderful thing, but it is a rarity in industry and only allows for correlation analysis to be done within the constrained scope of that integrated system. To consider correlations with events from an outside system puts us back in the same situation as beforehand. Given that most supercomputers are not completely single vendor supplied, this a substantial concern and reality for most monitoring endeavors. One possible solution would be to allow for an attribute in the log protocol specification that would be used to identify the event. Another solution would be by convention. This would have each message be prefixed with an integer or integer set that would identify the event. However, both of these would take years to propagate through industry and there would always be the cases of non-compliance.

One method born of necessity began as a way to filter log streams, is the use of a regular expression engine. This allowed systems administrators to email themselves or execute some script to deal with the occurrence of some event. Later, as organized storage of these occurrences became an evident need to prevent the re-processing of all the message strings for a data mining process, tagging was implemented. This, when combined with database indexing, solved one major issue, but left a number still outstanding. This method required a great deal of prior knowledge of every specific subsystem and its reporting methods. Given that there is always a new technology being incorporated with the next machine, domain expertise in advance is often a rare commodity. This also only allows things that have been specifically identified in advance to be tagged. Later identification processes involve a full re-processing traversal of the stored dataset, with reads and writes. This can become cumbersome. This system also allows for a great deal of information to slip through the cracks. If a regular expression does not catch all the possibilities, events will go handled. There is also the issue of maintaining the regular expression library. As software changes with versions, bug-fixes or driver updates, the regular expressions can become invalid and more events continue to be missed. But the true downfall of this system is that, by design, nothing happens to events that are not successfully identified. There is no way to gauge that something is happening on the system that an administrator should take note of if it has not already been specifically identified and coded for.

User againaru attempted to execute command: ps -a
User jfullop attempted to execute command: tar xvf archive_name.tar
User * attempted to execute command: * * n+

Table I
LOG TEMPLATE EXAMPLE

III. Hierarchical Event Log Organizer (HELO)

To address these issues, we have developed a novel, unsupervised classification tool named the Hierarchical Event Log Organizer (HELO), that aims to accurately determine events from log files. This tool takes sets of logs from any number of sources and uses statistical methods to derive a set of templates that can be used to describe the events contained therein. This allows all of the events that occur to be tagged with unique integer that will allow for enhanced mining in analysis later. It further handles events that do not directly match a template in the library by determining if the new message is very similar to an existing template and that the existing template should be slightly modified so the new message can be included, or if the message warrants the definition of a new template. The later would be the case when something new occurs that has not been seen before. HELO has to major modes. The first (Offline) builds the initial template library with high degree of accuracy, while the second (Online) tags and modifies the template library at a very high speed.

Events generated by the system are usually composed by two parts, the header and the message description part. The header has information about location and timstamp, like the first part in the following example *[2008-07-08 02:32:47][c1-0c1s5n0] 157 CMC Errors*. HELO analyses only the message description part and extract patterns by replacing the log variables, like the manipulated objects or states for the program, with wildcards. In our example, the template generated will be "*d+ CMC Errors*".

Our tool uses three types of wildcards: d+ represents numeric values, * represents any other single words, and n+ represents all columns of words that have a value for some of the messages and do not exist for others. In the example in Table I two types of wildcards are illustrated.

The extracted group templates are used to describe events generated by the supercomputers and to afterwards characterize the overall behavior of fault and failures in the system.

A. *Offline Methodology*

The offline component of HELO deals with mining group patterns from historic log file messages. Basically the algorithm groups events considering their description in a 2 step hierarchical process. In the first step the algorithm searches for the best split column for each cluster and in the second step the clusters are divided correspondingly. A split column represents a word position in the message description that is used to divide the cluster into different groups. An intuitive description of the methodology is described in Figure 1.

HELO starts with the whole unclustered log file as the first group and recursively partitions it until all groups have the cluster goodness over a specified threshold. The cluster goodness is a term used to characterize how similar all messages in one group are and is defined as the percentage of common words in all events' description over the average message length. The pseudo-code for this part is presented in Algorithm 1.

Algorithm 1 Offline Clustering Function

Name: Divide_cluster
Input: Collection G[] of partitions; cluster goodness threshold CT
Output: Collection G[] of partitions that have the goodness over the threshold
Function call from the main program: DivideCluster(G) where G is one cluster containing all events in the log file.

```
 1: Create null collection of partitions Gfinal
 2: for every partiotion in G do
 3:     Create null temporary collection of partitions Gaux
 4:     word_position = Find_split_token(G[])
 5:     Add to Gaux the partitions returned Split_in_clusters(G[], word_position)
 6:     for every partition in Gaux do
 7:         if Gaux[] goodness < CT then
 8:             Add to Gfinal the partitions returned by Divide_cluster(Gaux[])
 9:         end if
10:     end for
11: end for
12: return Gfinal
```

We will now follow the example presented in Figure 1 in order to explain the splitting process. The best splitting position is the one that contains the maximum number of constant words. We consider that words with a high number of appearances on one position have a higher chance of being a constant, so HELO searches for the column where most unique words have a high appearance rate. This position corresponds to the column where the mean number of appearances for every unique word is maximum while still having enough words in order to be relevant to the analysed event dataset.

HELO considers that different type of words have different priorities dependent on their semantics. There are three types of considered words: English words, numeric values and hybrid tokens (words that are composed of letters, numbers and symbols of any kind). The lowest priority is for all-numeric values since the algorithm considers that these words have the most chances of becoming variables in the clusters. In the example from above, 8 and 10 from column 2 represent the same token and so they are decreasing the appearance mean for all unique words. In our example the template "*Added d+ subnets and d+ addresses to DB*" comes from a group with 100% cluster goodness. Hybrid values are represented by tokens like "*check..0*" from our example. The algorithm extracts and considers only the English words incorporated in the hybrid token. For our example both "*check..0*" and "*check..1*" are considered as the word "*check*". If we analyze the second split from our example, the second and the third columns could be both chosen by the algorithm for the splitting position.

In the end, the group templates describe all types of events that the system has generated, in an intuitive way. The user-friendly group description generated by the tool could ease the work of system administrators to follow and understand errors from log files.

B. *Online Methodology*

The online clustering process deals with grouping messages in real time as they are being generated by the system. Clustering tools must be able to change the group templates in order to manage new messages that could appear. The input is given by the groups obtained with the offline process on the initial dataset. In HELO, each cluster needs to be represented by a description in the format described in the previous section and some statistics about the group.

For each new message, the online component checks the description of the messages and retrieves the most appropriate group templates. If a message fits the exact description of a group (this means the group template does not need to be modified) then the search is over and we stamp the message with the template's group id. If the message does not have an exact match with any of the groups then we compute the cluster goodness for all the clusters retrieved before, after including the new message in each of them. For computing the goodness of the group if the message is inserted, we retrieve the average length of all messages in this cluster from the group statistics file. This information is used to decide if including the new message decreases the cluster goodness under the threshold or not. If no cluster has the goodness over a specific threshold then a new group is formed. Else the group with the best cluster goodness will be chosen and the group template will be modified to accommodate the new message.

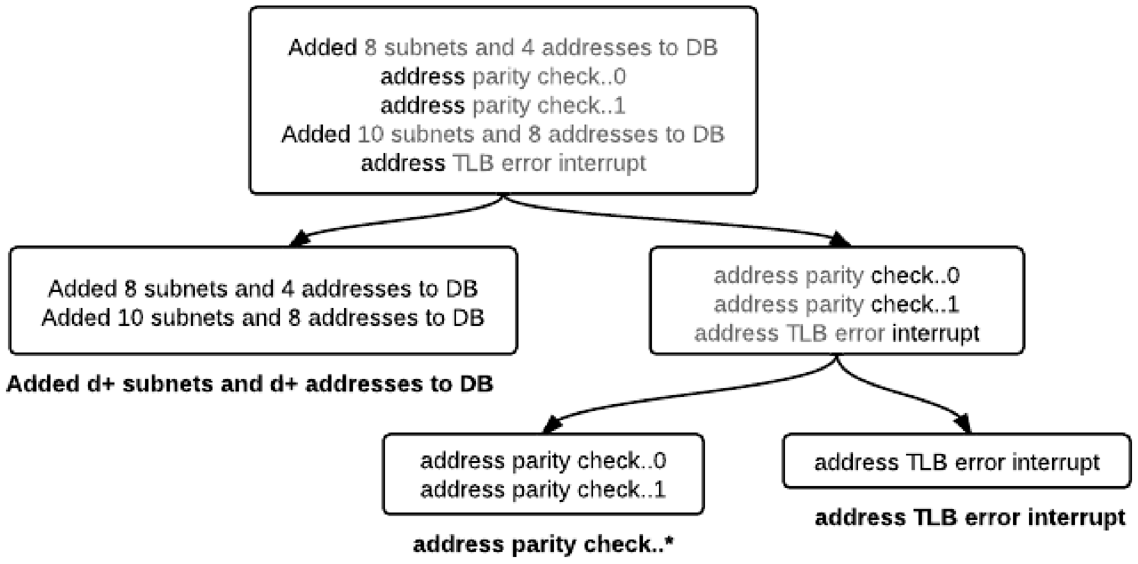

Added 8 subnets and 4 addresses to DB
address parity check..0
address parity check..1
Added 10 subnets and 8 addresses to DB
address TLB error interrupt
Added 8 subnets and 4 addresses to DB
Added 10 subnets and 8 addresses to DB
Added d+ subnets and d+ addresses to DB
address parity check..0
address parity check..1
address TLB error interrupt
address parity check..0
address parity check..1
address parity check..*
address TLB error interrupt
address TLB error interrupt

Figure 1. Online methodology

Algorithm 2 Online Clustering Function

Input: Event message M; Collection G[] of message groups; Collection Gstat[] of statistics information about each group CT as cluster goodness threshold
Output: The group id where the message belongs

```
 1: define null maxid and maxCG
 2: for every group in G do
 3:     sim = Compute_similarity(M, G[])
 4:     if sim == 100% then
 5:         return id of group G[]
 6:     end if
 7:     MT = Extract message templates from Gstat[]
 8:     Create temporary partition Gaux by adding M to MT
 9:     CG = compute_cluster_goodness(Gaux)
10:     if CG > CT then
11:         maxCG = CG
12:         maxid = id of group G[]
13:     end if
14: end for
15: if maxid is null then
16:     Create new cluster with description M
17:     return id for the new cluster
18: end if
19: Add information about M in cluster maxid
20: return maxid
```

The pseudo-code for this component is illustrated in Algorithm 2.

C. *HELO core summary*

HELO starts with an initial set of log files and generates the template library (offline). It then uses this library to classify and tag the output of the system so that it can be efficiently mined in future analysis processes.

IV. Additions to HELO

In order to run HELO on different systems, we implemented a couple of optimizations and additions that make the process applicable as a preprocessing step for a wide variety of analysis modules. First, we propose a parallelization scheme that makes the process efficient even in the absence of a powerful machine dedicated to analyzing the system. Secondly we added a new level in the analysis process by including the Source field. This allows us to have a component level analysis beside the existing system level one. Finally, we look at grouping the template list so that we can further analyze the events at a coarse or finer granularity. Each of the additions is described in the following subsections.

A. *Parallelization*

Our optimization was specific for IBM machines, that offer a set of service nodes that can be used for analysis purposes. However, this work can be applied on other machines as well, as long as we can choose a set of nodes where our analysys modules will not interfere with the application execution.

Since our method is almost embarrassingly parallel, we divide the online process on each service node offered by the system. The execution of each HELO instance will be independent one from the other and will need to interact only when a new message is generated. In our implementation we use a database on a centralized webserver to synchronize the list of templates from all the nodes.

The execution of the online component on each service node starts with the download of the entire set of templates extracted by using HELO on a centralized node. Each

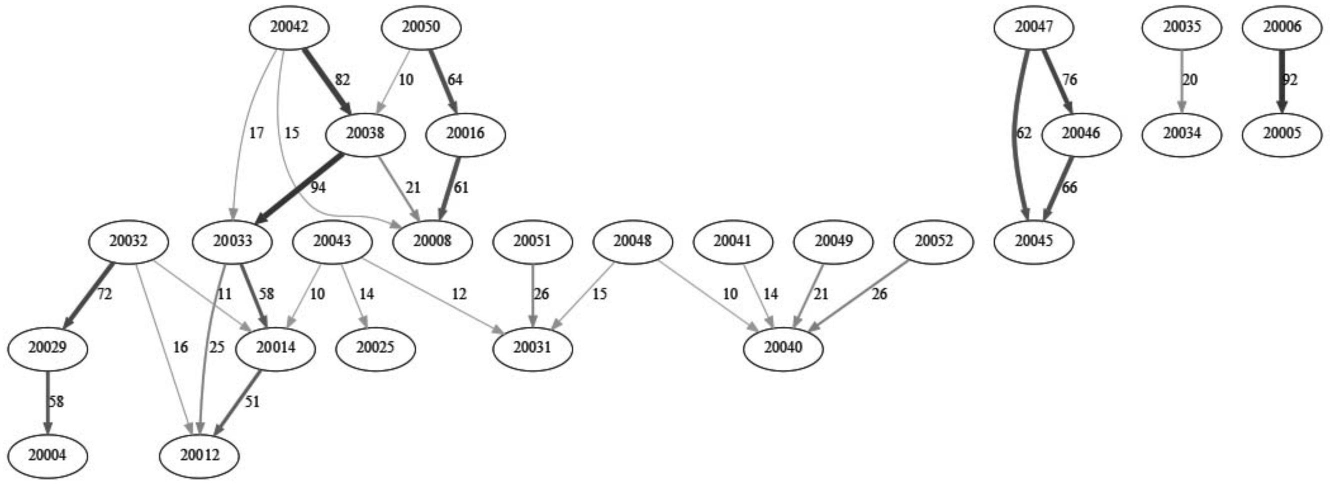


Figure 2. Example of a complex correlation graph

service node deals with an incoming stream of messages generated by the system. Each of them are transposed into event occurrences by matching the message to a template. The classification with an existing event type is done automatic and in parallel on each service node without any communication with any other nodes. However, in case of an unknown message the service node needs to communicate the modification in the template list, in order to update the centralized database. With this occasion, the service node downloads all the changes that occurred in the centralized set of templates since its last download.

The advantage of this approach is that each service node adapts its own list of templates to the output of the part of the system that it is analyzing. Even if the template lists are not consistent across service nodes, at all time, the centralized database holds the set of event types that describes the entire system. Another advantage is how the data is organized across the nodes, different modules can be added in the framework and execute on the event list in a pipeline manner, with the synchronization being guaranteed by the centralized database.

B. The Source field

The HELO tool was integrated into the fault tolerance framework for the Blue Waters system as a preprocessing module that represents an input to other analysis processes, like the correlation module presented in the next section. It is important to have as much flexibility as possible in the analysis modules. This is why is preferable to be able to change the granularity of the analysis dynamically depending on the demands of the prediction module [8].

We use the Source field to separate the events generated by different components in the system. This allows us to apply the correlation system separately on each component but also for the whole system. In practice, this is a major benefit as it allows for cross-system correlations to be found. For example, we can find when a network event causes storage system events.

C. Template grouping

We group the set of templates based on keywords from their description. By tuning the parameters in this process, we can change the granularity at which we characterize event types in the system. For example, for the lowest granularity, for the Blue Waters system, we obtain only 6 event types: network, memory, processor, disk and all others. However, when we tune the tool to give us the highest granularity, the template list consist of over 300 event types. These represent more specific events of the base 6 presented before. For example, *failed to configure resourcemgmt subsystem err = 10* still represents a processor message for both granularities, however has extra information when analyzed at a lower granularity. Specifically, in this case it represent a processor cache error.

V. Event correlation analysis

The correlation model takes as input the template list generated by HELO and its purpose is to find the set of events that frequently occur together. We use a brute force method by investigating all to all event types and extracting the pairs of correlating templates. We then use this information to generate a graph that describes the behavior of the system, a graph that can be used for prediction afterwards. In the following subsections we will describe in more details each step of the method.

A. Cluster analysis

The mining process is using DBScan [1] to decide when two event types are correlated. DBScan is a data clustering

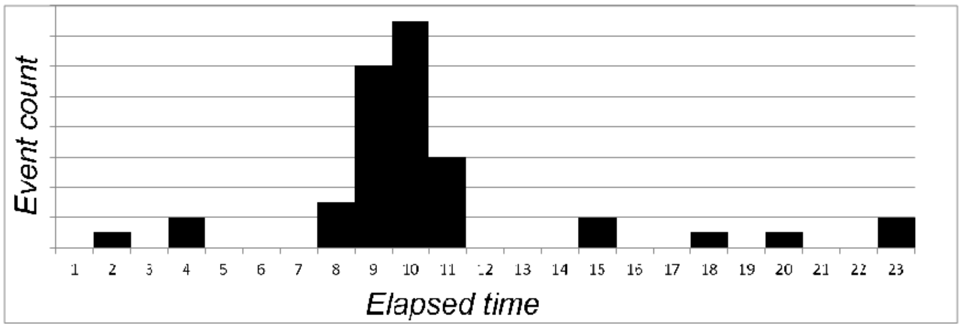


Figure 3. DBScan example of correlated events

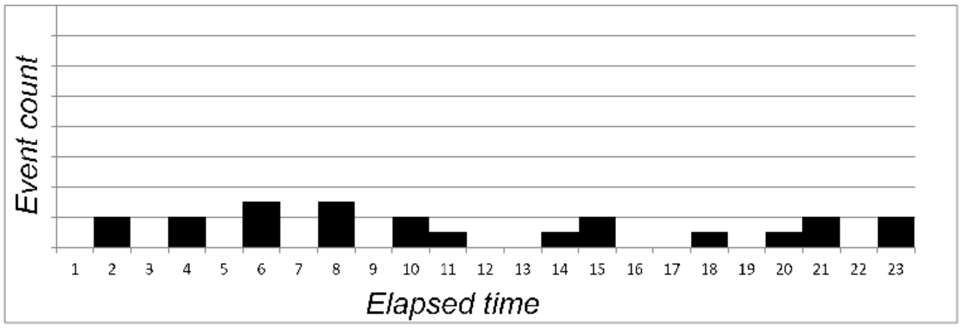


Figure 4. DBScan example of uncorrelated events

algorithm that is usually used to find all clusters that have an estimated density distribution. In our implementation DBScan uses only one input parameter that influences the results, specifically the minimum number of points required to form a cluster.

For each event type A, we use DBScan by looking at all preceding events of type B within a window and recording the number of seconds between them into a set. We observed that for previous systems the number of correlated events separated by a time delay of more than a couple of days is very low. For this reason we chose a maximum time window of one week, to limit the search for DBScan and make it more efficient. The constructed set for the correlation $\vec{BA}$ is processed to become the input array for DBScan. This process consists of counting the number of occurrences of each time delay. The pseudo-code is presented in Algorithm 3.

In statistics, a result is called statistically significant if it is unlikely to have occurred by chance. We use the Pearson's significance test with two tailed t-values [3] to determine the threshold over which our correlation has statistical significance. The significance test for Pearson's r is computed as follows:

$t = \frac{r*\sqrt{N-2}}{\sqrt{1-r^2}}$

with r is Pearson's correlation value, t is the two-tailed probability value and N is the number of samples that went into the computation of r. From a t table, by taking into consideration the number of samples that create the signal, the two-tailed probability value can be found. So, we can use the previous equation for finding the minimum correlation value that would still give statistical significance for our signal:

$r = \frac{t}{\sqrt{t^2+N-2}}$

For example, if we are analyzing the correlation between two events A and B, and the number of occurrences of the less frequent event is 50,000, this means t will be 164. By using the second equation, we will get a r-value of 0.61. This means that a peak from DBScan must have at least 61% of total delays in order to represent a correlation. This is the value that we use as a threshold for determining the minimum number of points required to form a cluster.

Figure 3 and figure 4 present two examples of processed sets of time delays. The first one shows a peak for time delays between 9 and 11 time units. This means that this time delay is very frequent in the log file and indicates a strong correlation between those event types. It also gives an average time delay of 10.6 minutes unites with a standard deviation of 0.86 minutes. The second figure does not show any peaks, which indicates a lack of correlation between the event types. In both figures, the lower valued peaks are

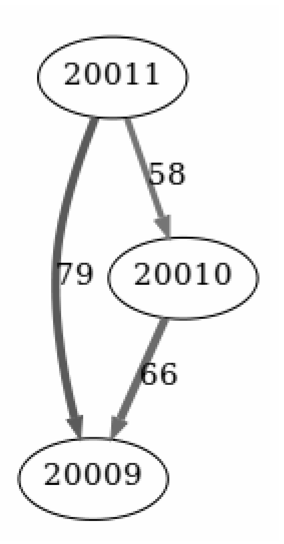


Figure 5. Example of correlation graph

filtered out by the input threshold.

One of the main advantages that DBScan has over other mining techniques [5] is that it does not require the number of clusters a priori. Also, it does not require a filtering step that related work relies on, which is known to decrease the accuracy of the results [4], [10].

In its original form DBScan visits each point of the database multiple times and results in a complexity of O(n*log n). However, by taking into consideration the behavior of HPC events we optimised our algorithm to run in O(n). The service nodes running this module handle the task easily, so there is no problem executing the all to all computations, even if most of the pairs are not correlated. This method guarantees that we extract all of the correlations in the system.

B. Generating graphs

The correlation module records the average time delay and the standard deviation between all the correlated event types. We then parse the log file to further characterize the correlations by extracting the count and the confidence. The confidence value represents how many times the correlation takes place divided by the number of occurrences second event.

$$conf(\vec{BA}) = \frac{count(\vec{BA})}{count(A)}$$

With this information we construct a graph that describes the behavior of the system and that will be used for event prediction.

To make things more clear, we will provide a couple of examples of correlated chains extracted from the graph after analyzing a BlueGene machine. In the first example the chain is composed of only informational messages and is a sequence of messages generated each time the system controller restarts.

”*starting systemcontroller*”
after just a couple of seconds:
”*controlling bg/l rows [0 1 2 3 * n+*”
after one minute and 30 seconds:
”*running as background command*”

This sequence of events is extracted from the graph presented in Figure 5 after applying DBScan on all to all combinations between the event types provided by HELO. The nodes are represented by template ids and the links show the confidence value between each event type. The time delay between the events is given by the peak indicated in DBScan. For example between the last two event types presented previously, DBScan showed a peak corresponding to 30 seconds, with a very low standard deviation.

A more complex graph could indicate event more complex correlations. Figure 2, at the end of this paper, illustrates a graph extracted from one of the LANL HPC system [6]. The following example, presented in Table II presents one of the chains found in this graph which contains FAILURE messages in the sequence of notifications.

The correlation graph will be analyzed into more detail and used for prediction in the next section.

C. Special considerations

1) Fan-in/Fan-out: There are two interesting scenarios that are not caught by any other data mining algorithm apart from DBScan. Specifically, this is the case of many to one patterns, fan-out and fan-in. The many to one fan-out pattern occurs when a single event type precedes multiple events that are identical between them. For example, if the log frequently experiences the sequence of events $B\vec{A}A$, this represents a two to one fan out pattern. The many to one fan-in pattern is represented by multiple identical events preceding a separate event. This is the case of $B\vec{B}A$ patterns.

These patterns put other data mining algorithms in difficulty, however with DBScan they are easily identified by multiple peaks in the DBScan set. For example, if figure 3 would present two peaks, we would have a two to one pattern.

Large scale systems experience a large variety of events during their lifetime and they output notifications for each of them. Once an error is triggered, there is not a consistent way of registering how the system will behave. For example, in case a node experience a network failure and is incapable of generating log messages, the failure is announced in the log files by a lack of generated messages. Conversely, some component failures may cause logging a large numbers of notifications. For example, memory failures can result in a single faulty component generating hundreds or thousands of messages in less than a day.

** is not fully functional*
after 27416 seconds (7 hours)
warning no ethernet link
after 7019 seconds (2 hours)
*rts tree/torus link training failed: wanted: * n+*
after 24185 seconds (6 hours)
job d+ timed out. block freed.
after 20 seconds:
ciodb exited abnormally due to signal: aborted
*mmcs_server exited abnormally due to signal: * n+*
after less than 10 seconds:
*mmcs_db_server has been started: ./mmcs_db_server –usedatabase bgl –dbproperties * –iolog /bgl/bluelight/logs/bgl –reconnect-blocks all n+* (this is probably a restart due to the failure)
ciodb has been restarted.

Table II
SEQUENCE OF CORRELATED EVENTS

Algorithm 3 Correlation extraction

Input: List of templates Template T[], the Log file L;
Output: List of correlated pairs of events Corr[] and corresponding list of time delays for each of them D[] (Coor[i]=(A B) with a time delay between them of D[i]=(avg_time, std_dev))

```
1:  min_pt = Pearson_threshold(count(A),count(B))
2:  for every template A in T do
3:      for every template B!=A in T do
4:          delay = Extract_delays(A, B, log)
5:          dbarray = Extract_count_for_each_delay(delay)
6:          CG = DBScan(dbarray, min_pnt)
7:          if len(CG) > 0 then
8:              Add (A,B) to Corr
9:              Add CG to D
10:         end if
11:     end for
12: end for
13: Function DBScan(dbarray,min_pnt):
14: CG = []
15: for every delay in dbarray do
16:     if dbarray[delay] < min_pnt then
17:         Filter noise
18:         if start is set then
19:             Add (delay+start/2, delay-start/2) to CG
20:         end if
21:     else
22:         start = delay
23:     end if
24: end for
25: return CG
```

2) Periodic events: Event types exhibit three types of behavior: periodic, silent and noise [7]. In our procedure we analyze periodic event differently than the others. Usually, periodic signals are generated by daemons or by events that deal with monitoring information, like cron. The reason why these type of signals need to be analyzed separately is because their behavior creates extraneous correlations with other periodic events, and in DBScan multiple peaks would be found. These peaks could be mistaken for the fan-out pattern.

3) Same-source accuracy: Even though the ability to find correlations across sources is a phenomenal benefit, it does not come without a cost. As machines grow in node counts, so does the probability that another node somewhere will have an event that occurs in the window of time preceding the event found in the mining process while building the elapsed time sets for cluster analysis. In fact, something could easily cause many nodes to report similar syslog messages at nearly the same time. When node time skew is added to the situation, the data set for cluster analysis becomes very noisy if we consider the entire domain for preceding events. To address this, if the trailing event and preceding events are both from the same source, we add the further qualification that their locations should match in our mining process. This significantly cuts down the size of the data set used for cluster analysis while greatly improving the accuracy of the correlation.

VI. PREDICTION

Our mining and analysis operations generate an event relationship table that tracks the two events in order of temporal relationship, average time between them, the standard deviation of that relationship and the probability of the following event occurring given that the preceding one has, and the count of the following event cluster. The average time between events allows us to determine how far in the future a predicted event may occur. The standard deviation is used to determine the window in time to expect the future occurrence. The probability helps us determine the chance that the event may occur. Much of this should be self evident, but needs to be stated for the explanation of chains of events.

A chain is a series of related events that can have its predictive attributes calculated in the following manner. The estimated time of occurrence of the final event in the chain from the first is simply the sum of the average time values for each relationship contained in the chain. The window can be described as plus or minus the sum of some surety

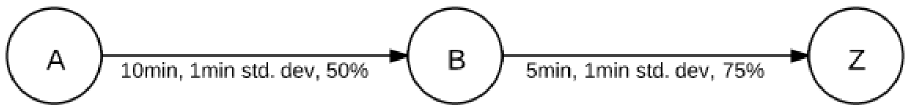


Figure 6. Event chain example

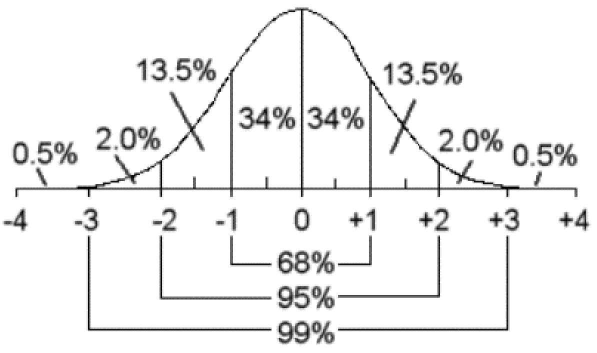


Figure 7. Normal distribution curve

constant multiplied by the standard deviation of average times for each relationship contained in the chain. For our implementation, we have chosen to use a surety constant of 2. Utilizing a normal distribution curve, two standard deviations greater and two standard deviations less than the average value should encompass 95% of the expected results. (Figure 7 [2]) A smaller factor results in a smaller window, but also a lower rate of prediction success. This is an engineered feature that can be used to tune the prediction engine to its area of application. Finally, the probability of the final event occurring is the product of the series of probabilities of the events in the chain.

For clarity, we offer the following example for a chain of events $A->B->Z$. ($A\vec{B}Z$) from Figure 6.

Probability of Z = $P(A|B) * P(B|Z)$
Time until Z = $avg.time(\vec{AB}) + avg.time(\vec{BZ})$
Window for Z = $2 * std.dev(\vec{AB}) + 2 * std.dev(\vec{BZ})$
Calculations:
If Event A occurs at 1335675600 unixtime.
Probability of Z = 50% *75% = 37.5%
Time until Z = (10+5)*60 (seconds) = 900 seconds from A (@1335676500)
Window for Z = (2*1 +2*1) *60(seconds) = 240 seconds

We can predict that there is a 37.5% chance that Z will occur at 1335676500 with a window of +/- 240 seconds. If B occurs, the prediction would change to 5 minutes from the occurrence of B +/- 2 min with a probability of 75%.

A. *Prediction Expiration*

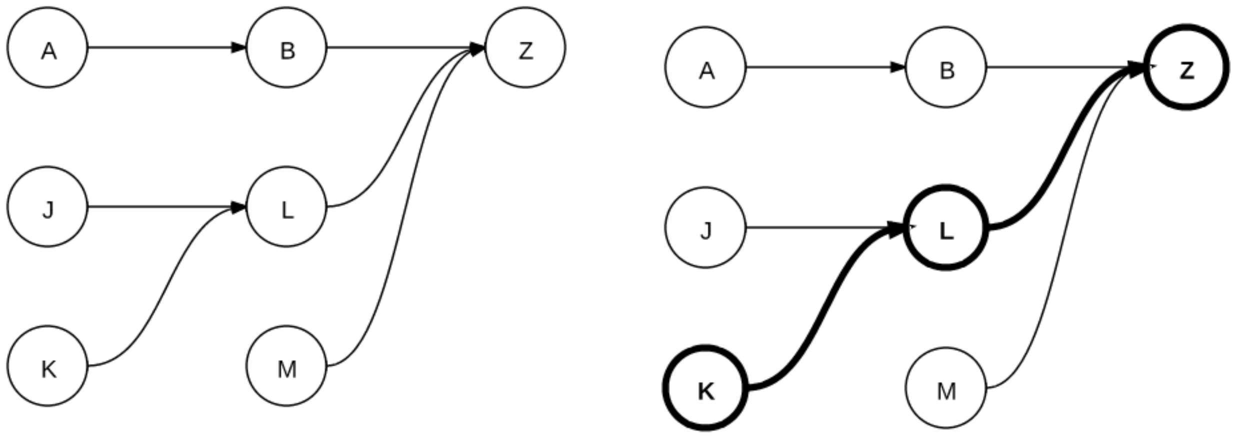
If an intermediate event fails to occur within (avg. time + 3 * std. dev) seconds from the occurrence of its predecessor, the prediction expires. In our example $A\vec{B}Z$, if B does not occur within 13 minutes of A, the prediction expires. This poses the question 'What if B ends up occurring after the prediction has expired?' An algorithm could continue to actively watch for the occurrence of B, but in practice that takes a significant number of cycles for a computer to accomplish given the extreme potential of predictions that could arise in a short period of time. This further begs the question of what methodology should be used to analyze large sets of real-time data to find and react to these chains of events.

B. *Implementation*

Since our relationship table can be used to generate a graph (Figure 8(a)), which is in essence a set of interwoven chains leading to an Event of Interest (EoI) (Figure 8(b)). We can use these chains and the methods described above to predict the future occurrence of an EoI.

The first and most obvious option would be to watch for the occurrence of a preceding event, starting at the originating end of the chain. This is a valid method, but as chains grow in length, so does computational cost. This is because any subset of the chain is also a valid chain. And since each event in the chain could occur independently without any preceding events, this becomes complex. Bayesian network generation is an option. But since we are working with purely observed data and no insight into the inner mechanisms of the system we are monitoring, accurately determining conditional situations from the raw data would be a difficult task. Furthermore, there is the issue on how difficult it will be to keep that network updated with a constant flow of new data.

Our approach is to first identify each EoI that is to be monitored for prediction. For each EoI, we recursively traverse the graph backwards from the EoI until we find an event that has occurred within the cumulative time plus window from now; we reach a maximum cumulative time depth or steps; or we reach an event node with no predecessors. This gives a number of benefits. First, the most recent event, graph-wise, is the most important. Events before the preceding event that has occurred have little value in the given linear chain. Remember that it is possible for an event to be a preceding event for multiple following events,



(a) Graph of events

(b) An example of interwoven chain

Figure 8. Example of graph of events and its corresponding chains

of which there might be a relationship. (e.g. $\vec{AB}$, $\vec{BC}$ and $\vec{AC}$). Those relationships will be considered independently by the recursive graph traversal. A second benefit is that we can easily control and enforce boundary conditions to manage computational load with the maximum windows, number of steps, and/or total elapsed time. Additionally, we do not have to pre-determine chains or find them in the graph each time we wish to ascertain if there is a chance for an EoI.

The above discourse has shown how we are able to predict what event will occur when and with what probability. The remaining component for predicting these lightning strikes is determining where they will occur. With the current location reporting facilities of syslog, it is fairly direct that we can determine the location of a future occurrence in that for a node, a preceding event occurrence's location would determine the location of the trailing occurrence. This works well enough with event chains that are wholly contained within the scope of a node. However, one of the great features of this system is the ability to correlate across sources or subsystems (i.e. networking to storage system). In the cross-source events, the best that we can determine in terms of location is which subsystem, and not necessarily the specific host resource within a particular subsystem.

VII. Future research

Topology awareness - This is an area of research that could provide an enhanced location component of predictions. We envision a database housed relation mapping for physical connections as well as configuration adjacencies. The major areas of impact would be in the expanded description and consideration of the mining techniques to include the topology map traversal.

Event Fingerprinting - Oftentimes an event takes place that does not have a distinct syslog message to indicate that something has occurred. However, it manifests itself in the logs as a pattern of events. We are aware of a couple of projects that finds event sets or patterns and distills them down into a singular event. These techniques could be incorporated into our system in the future.

Root Cause Analysis - Given our event relationship graph, we have seen how we can predict future events focused on the end of the chain. However, we are interested in seeing how the origin end of the chain can be used for determining root causes of events. We look forward to determining the implementation of algorithms to provide usable research tools to systems administrators.

VIII. Conclusion

With the world of leading edge computing systems constantly changing, trying to maintain libraries and rule sets to handle the massive amount of data reported by these machines is daunting task to do and virtually impossible to do well. The thing we realized is that we'll rarely ever know what to look for going into the building of a new supercomputer. Once we came to terms with that, we began to look at the problem from a different perspective. We created an intelligent, learning system that did not let things fall through the cracks. Being able to distinguish the difference between events and their occurrences and organize them in such a way that we could efficiently analyze them made the correlation an engineering possibility. Visualizing the relationships in graph form led us to the algorithms for prediction. Along the way we found a number of coincidental benefits that in sum have the potential to change the way supercomputers are monitored and managed. We have also tried to keep the structure fairly generic so that it may be applied in other areas of analysis beyond supercomputer monitoring. We hope that this provides a step in the right direction and look forward to finding improvements in the future.

References

[1] Martin Ester, Hans-Peter Kriegel, Jrg Sander, Xiaowei Xu A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise in Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96) Volume 2 Issue 2, Pages 169-194, June 1998

[2] AllPsyche Online http:// allpsych.com/researchmethods /distributions.html Accessed on April, 2012.

[3] H. Posten: The robustness of the twosample ttest over the Pearson system. Journal of Statistical Computation and Simulation, Volume 6, Issue 3-4, 1978

[4] A. Pecchia et al: Improving Log-Based Field Failure Data Analysis of Multi-Node Computing Systems DNS 2011

[5] N. Nakka et al: Predicting Node Failure in High Performance Computing Systems from Failure and Usage Logs IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems, 2011

[6] Bianca Schroeder and Garth A. Gibson. A large scale study of failures in high-performance-computing systems. International Symposium on Dependable Systems and Networks (DSN 2006).

[7] Ana Gainaru, Franck Cappello, William Kramer Taming of the Shrew: Modeling the Normal and Faulty Behavior of Large-scale HPC Systems. 26th IEEE International Parallel & Distributed Processing Symposium 2012

[8] Z. Lan and all. Toward automated anomaly identification in large-scale systems. IEEE Trans. on Parallel and Distributed Systems, Volume 21, Issue 2, Pages 174187, February 2010.

[9] Zhiling Lan and all Enhancing Application Robustness through Adaptive Fault Tolerance NSFNGS Workshop (in conjunction with IPDPS) Pages 1-5, April 2008

[10] A. Pecchia et al: Improving Log-Based Field Failure Data Analysis of Multi-Node Computing Systems 41st International Conference on Dependable Systems & Networks (DSN), Pages 97-108, 2011