Real-Time System Log Monitoring/Analytics Framework

Raghul Gunasekaran, Sarp Oral, David Dillow, Byung Park, Galen Shipman, Al Geist
Oak Ridge National Laboratory
{gunasekaranr,oralhs,dillowda,parkph,gshipman,gst}@ornl.gov

Abstract

Analyzing system logs provides useful insights for identifying system/application anomalies and helps in better usage of system resources. Nevertheless, it is simply not practical to scan through the raw log messages on a regular basis for large-scale systems. First, the sheer volume of unstructured log messages affects the readability, and secondly correlating the log messages to system events is a daunting task. These factors limit large-scale system logs primarily for generating alerts on known system events, and post-mortem diagnosis for identifying previously unknown system events that impacted the systems performance. In this paper, we describe a log monitoring framework that enables prompt analysis of system events in real-time. Our web-based framework provides a summarized view of console, netwatch, consumer, and apsched logs in real-time. The logs are parsed and processed to generate views of applications, message types, individual/group of compute nodes, and in sections of the compute platform. Also from past application runs we build a statistical profile of user/application characteristics with respect to known system events, recoverable/non-recoverable error messages and resources utilized. The web-based tool is being developed for Jaguar XT5 at the Oak Ridge Leadership Computing facility.

1 Introduction

System logs, generally referred to as RAS (Reliability, Availability and Serviceability) logs, provide a wealth of information on the status of large scale computing systems. Often these system logs are large volumes of unstructured and redundant information, which affect the readability and easy interpretation. The Jaguar XT5 typically generates a few hundred thousand log messages per day. Moreover, interpreting these logs for diagnosing system or application anomalies requires an extensive understanding of the system state and environment. System administrators are tasked with the tedious and daunting effort of isolating problem and understanding system failures from logs. Though a number of commercial log analysis tools such as Splunk are being adopted at a huge cost, they simply serve as a sophisticated query interface. These factors limit the usage of the logs on a regular basis and are primarily used only for diagnosing previously unknown system events that cause serious performance degradation. This results in a number of silent errors going undetected, which can result in poor application performance.

System log and metrics are time stamped values collected regularly over time and aggregated over the entire machine. In general the log messages can be parsed to identify the source generating the error, the error type, the error message, and the potential target (entity at fault) based on the type of error. Combining the RAS and scheduler logs allows us to associate errors with measurements to examine a specific application that was running during the

This research used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725.

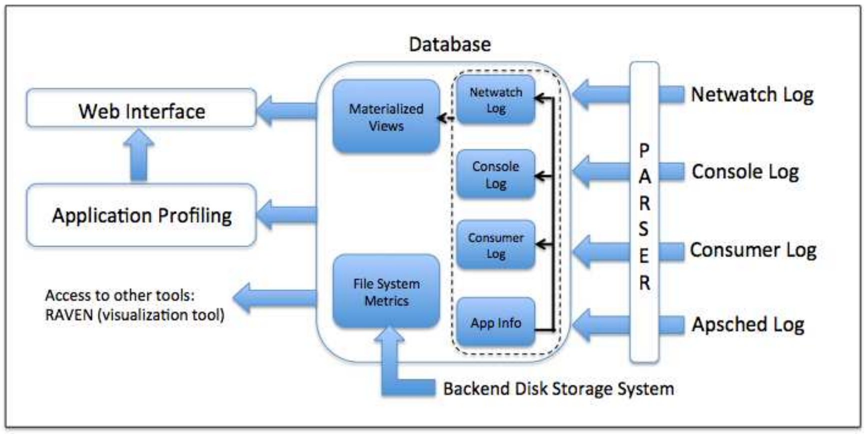

Figure 1: Log Monitoring/Analysis Framework

time period. With an understanding of the compute infrastructure, we can further categorize the messages based on source/target types, physical location and mapping/dependence between the various entities. Associating these information points with each other, we can generate summarized views of log messages in a more precise and readable format. Furthermore, by observing multiple runs of applications over longer period, we can build a profile for individual applications based on observed system events and metrics. In comparison to traditional tools, our approach of profiling does not provide fine-grained details of application behavior, but simply model the runtime characteristics of the application. Also, the general profiling tools [6, 2, 1, 14] have a significant compute overhead (4-8%) and bandwidth requirements while capturing trace information, so they carry an unacceptable cost when used for continuous monitoring of applications. Our approach of profiling will help model typical application runs in terms of system events, which can be used for understanding the resource utilization of individual applications and anomaly detection. We define anomaly as a deviation from the expected behavior of an application. This does not necessarily imply that application is at fault, but the abnormal behavior of an application could indicate shared resource constraints or impact by other applications sharing the same platform. Our approach of characterization and anomaly detection are indicators of poor or undesirable application performance and act as triggers for further investigation.

In summary, we propose a framework for processing/analyzing log messages and enabling summarized views of log messages via a web interface. First, we process raw log message streams by structuring the log messages and direct them in to MySQL database tables. Second, we generate materialized views of log messages with respect to application, error-type, physical location (cabinets, nodes) and source-destination pairing. Third, we enable querying/viewing log messages via a web interface in real-time, and also allow other users/tools to query raw log from the database. Finally, we use system log and other system metrics to profile the runtime characteristics of individual applications. Our implementation is based on the Oak Ridge Leadership Computing Facility (OLCF) Jaguar XT5 system logs and Spider center-wide file system [13] backend disk stats. The framework described in this paper extends our previous work described in [10].

2 Monitoring Framework

To enhance the readability and ability to analyze logs in real-time, we are developing a log monitoring framework for the Jaguar XT5, illustrated in Figure 1. Organizing the log messages is a two step process, first *pre-processing* where the raw logs are parsed and stored in database tables. Second, the log entries are grouped by associating messages within a time window to provide a concise summary of the log messages, referred to as materialized views. The monitoring framework is being developed on a dedicated server hosting a MySQL database and apache webserver. The log streams are directed to

Table 1: A few examples of RAS and Apsched Log

Netwatch Log

Timestamp	Node ID	Port	Remote Node	Port	Errs
100130 11:52:39	c4-6c2s0s0	5	c4-4c0s0s3	0	1 uPacket Squash
100130 11:52:39	c7-5c1s0s0	5	c7-7c1s0s3	0	3 uPacket Squash
100130 11:52:39	c23-2c1s0s0	6	*	*	Deadlock Timeout, Req Chan

Console Log

```
[2010-06-11 00:24:21][c14-2c0s4n0]beer: cpu_id 9: nid 9812, cpu 0 has been unresponsive for 240 seconds
[2010-06-11 00:24:21][c14-2c0s4n0]LustreError: 773:0:(ptllnd_tx.c:469:kptllnd_tx_callback()) Portals error
    to 12345-9812@ptl1: PTL_EVENT_SEND_END(9) tx=ffff8103f99bcd80 fail=PTL_NAL_FAILED(4)
    unlinked=0 1276230767 ref 2 fl Rpc:N/0/0 rc 0/0
[2010-06-11 00:24:21][c14-2c0s4n0]Lustre: widow1-OST0015-osc-ffff8103f9e8d000: Connection to service
    widow1-OST0015 via nid 10.36.227.118@o2ib was lost; in progress operations
    this service will wait for recovery to complete.
[2010-06-11 00:24:23][c16-2c0s6n1]HARDWARE ERROR
[2010-06-11 00:24:23][c16-2c0s6n1]CPU 0: Machine Check Exception: 0 Bank 4: dc04400040080813
[2010-06-11 00:24:23][c16-2c0s6n1]TSC 4b9ec89ff4f6 ADDR 8e497800 MISC c0090fff01000000
```

Apsched Log

```
01:20:05: Confirmed apid 4325785 resId 349 pagg 0 nids: 12706,12710,13446,13506,13510,13696,13700,...
01:20:06: Bound apid 0 resId 349 pagg 16574 batch 418808
01:20:06: Placed apid 4325786 resId 349 pagg 16574 uid 63137 cmd jobcleanup nids: 12706,12710,...
02:14:23: Released apid 4325786 resId 349 pagg 16574 claim
02:14:23: Canceled apid 4325785 resId 349 pagg 16574
```

the server via *syslog-ng* from the Cray SMW. The file system metrics are queried periodically from the back-end storage system, described in [11].

2.1 Log pre-processing

The Jaguar XT5 system status is monitored via four log streams, *syslog, console, netwatch*, and *consumer* log. For our analysis we analyze only the last three log streams, which are generally referred to as the RAS logs, while the *syslog* is not used in our analysis. We also study the *apsched* log from the ALPS (Application Level Placement Scheduler) subsystem, the Cray supported system for placing and launching applications on the XT nodes.

To improve the readability without loss of information, we parse the log streams and store them in MySQL database tables, each log stream is stored in a separate database table. The RAS log entries are time stamped values and are generated by a specific node, a few sample log entries are shown in Table 1. From these log messages we can parse out the `timestamp, source, target`, and `error type`, which uniquely identify every log entry. Source refers to the node generating the error and target refers to the entity (node, router, OSS) the source is complaining about. The verbose description of the error in the log message is also stored in the tables as `error message` for supporting more detailed views, described in the next subsection. The *netwatch* and *consumer* log have a more definitive structure. For example, for the *netwatch* log shown in Table 1, all messages have the same structure with different error types. However, the *console* log messages are more verbose and unstructured, and reports on Lustre (file system), BEER, machine check exceptions, out of memory, are a few of them. Identifying the target information from the *console* logs is not straight forward as the target information is embedded in the error message, and the structure of the message varies based on the error type. This gets more complicated for Lustre error messages, as extracting the target information depends on the specific error message. For handling Lustre error messages we parsed CDEBUG messages from the source code categorized as

D_EMERG and D_ERROR. We then defined regular expressions for each individual message identifying the various components of information [10]. All the regular expressions are stored as a text file and are accessed by the parser. Each log stream has a separate parser, written is python, which uses regular expressions to match log messages, parses the log, and writes to the database tables.

The *apsched* log records the allocation of nodes for a specific user job. The log have a well defined structure, as documented in [7], as shown in Table 1. The log entry with keyword *Confirmed* and *Bound* record the allocation of compute nodes to be a specific user job identified by a application ID (*apid*), reservation ID (*resId*), and session ID (*pagg*), which uniquely identify an allocation. Log entry with keyword *Canceled* marks the end of the job. Entries with keyword *Placed* and *Released* mark the usage of nodes for appruns initiated by the user within the job, which is identified by a new *apid* and the same reservation and session ID as the job. For identifying a node to the scheduled application we simply the use *apid* associated with individual apprun.

Each raw RAS log entry, as described above, is parsed to identify `timestamp, source, target, error type` and `error message`, where each entry is a table column. To facilitate further analysis of the logs the following fields (columns in the table) are also inserted into the tables in real-time.

- `appID`, from the *apsched* log we can associate every node to an application ID or *appID*. This information is added to every RAS log entry identifying the application the source node is running.

- `sourceType, targetType`, we identity very source or target entity as a compute node, I/O node, service node, router, OST, OSS, MDT or MDS.

- `sourceID, targetID`, in general the source node is labeled using CID (example c14-2c4s5n3), which identifies the nodes physical location in the cabinet. The target information in the logs are represented as CID, IP address, NID or hostname. NID is an integer value identifying the nodes on the compute platform. To facilitate analysis we maintain a uniform identification schema, where all nodes on the compute platform are identified by NIDs and all other entities are prefixed by their type, such as `ost, mdt, oss, mds or ib` followed by an integer value, example `ost432`.

Apart from the `appID-node` mapping, the mappings described above are fixed and specific to the OLCF compute infrastructure. The mappings are encoded as a configuration file and made available to the parser.

Table 2: Aggregated Log Messages

Console Log
Time: 05:10:00 05:20:00
Applications: 49
Source: 3427 SourceType: cnode
Target: 1 TargetType: rtr
Target Node: 6311
Lustre PTL errors: 10194
Netwatch Log
Time: 12:34:00 12:44:00
Applications: 17
Source: 1183 SourceType: cnode
Target: 1136 TargetType: cnode
Column: 1,2,3,4,5.24
Row: 5
upacket squash: 12115
Deadlock Timeout: 3542
Appid: 26457
Starttime: 18:34:00
Nodes: 2000
Source: 347 SourceType: cnode
Target: 34 TargetType: cnode, rtr
BEER errors: 234
MCE: 37
Lustre PTL errors: 592

2.2 Log views

Having organized the data in a structured and queryable format, we are still overwhelmed with the volume of information. Correlating these messages to the systems state and environment we are able to generate concise summarizes of the log. By system state, we mean details of node allocations for application and condition of system components (active/failed). System environment refers to the properties and dependencies of various hardware components, such as the mapping between OST, OSS and RAID controllers and individual nodes types (compute, I/O, service) on the compute platform.

To generate materialized views, which are precise overview of multiple lines of log entries, we cluster the log messages in two different ways. One, we cluster log messages over the runtime of the application, and secondly, we cluster log messages in fixed time widows. The first approach will help view events that are occurring on the nodes running the same application, and the second approach will help identify errors related to specific hardware and are not related to any specific application.

Table 2 shows examples of summarized view of log messages. The *console* log message shows 3427 distinct source nodes complaining on a single router node generating 10194 portal error messages within a ten minute period, a total of 49 applications are affected. In the second example, the grouping of the *netwatch* log, we see heavy network congestion on cabinets in row five, which implies congestion along a specific axis in a segment of the 3D Torus. This congestion affectings 17 different applications. Finally, when clustering with respect to application, we see the errors generated by an application running on 2000 nodes since its start time. This application generated Lustre portal errors, BEER, and MCE (machine check exception) errors.

The clustering approach was implemented within the mysql database with the help of *sql events* and *procedures*. To enable materialized views we create additional tables in the database that store aggregate count of errors based on applications and for individual logs streams. For aggregation based on a application, a table(`app_log_view`) indexed in `timestamp` and `appid`, maintains a list of source and error types and a count of the errors being reported for each type. A *sql event* is timed to execute a *sql procedure* periodically in one minute intervals. The *sql procedure* basically aggregates data from all the RAS log tables generated in the last minute and updates the `app_log_view` table. Similarly, for the RAS logs we maintain individual aggregation table, which are also updated in one minute granularities. Also, we aggregate Lustre messages from the *console* log in a separate table. As described in earlier section, we have extensively defined Lustre error messages as regular expressions, which help present more detailed views of the logs. This helps us identify errors with respect to the module within Lustre, such as *lnet, ptl, llite, ldlm, obd, or mdc*, that is causing the error. Though the aggregation within the database is limited to one minute intervals, such organization of data allows querying at lower granularity, say ten minute interval, through simple sql queries, which execute in few tens of milliseconds.

2.3 Other System Metrics

Apart from the system logs, we collect file system usage metrics from the back-end disk storage system. The *Spider* [13] file system has 96 DDN RAID controllers, which are accessed by the compute nodes via the Object Storage Servers. We monitor the read/write bandwidth and IOPS usage, tier delay and request size distribution from all the RAID controllers. This information is used to actively monitor the file system usage in real-time. We use this information to model the file system usage of individual applications, detailed in the next section.

Organizing the raw data and correlating them is done within the database, which can be queried by external users. However, the benefit of this work is in presenting such correlated information via a web interface in real-time and in a usable format for system administrators. We have a basic web interface to view the logs and metrics in real-time. We are currently working on presenting this information via a web interface in an interactive manner for system administrators.

3 Profiling Applications

Profiling helps understand the functional and resource utilization characteristics of individual applications. In general, profiling is done using fine-grained performance tools and is undertaken by application developers in collaboration with performance experts for optimization purposes. However, the runtime behavior of application on shared resource environment is very different from that observed on controlled test environments. Also, it is important to profile individual user-application behavior rather than grouping users with a common scientific application. Users, based on their scientific needs, define various parameters at runtime, such as the number of compute nodes, dimensions of the compute grid, I/O usage (frequency of checkpointing), all of which determine the true runtime characteristic of an application. In our observation of the jobs submitted to the Jaguar XT5, every user has one or more fixed job allocation models. In this section we present a few preliminary findings on profiling user application based on *netwatch* log and

I/O usage. The analysis presented in this section is based on four months of data collected between June and September of 2010. Our approach of profiling applications based on logs and system metrics can help identify application's resource needs and provision system resources accordingly.

3.1 Network Utilization

The *netwatch* log reports on the link status of the SeaStar 3D torus interconnect. A few sample log entries are shown in Table 1. The first field is the timestamp of the log entry, followed by the source and remote node IDs and port numbers. Every SeaStar router has 6 ports (numbered 0 to 5) and is connected to six neighbor nodes. The last two fields are the number of packet squashes and the error type, read as *micro-packet squashes.* The packet squash error indicates the number of retransmissions required on a specific link, which is sampled every minute independently on each SeaStar router. Log messages are generated when a nonzero number of packet squashes are detected in the sample interval. The packet squash error, being a data link layer message, can indicate data loss, data corruption or simply a bad link (hardware problem). Hardware problems were ruled out in our analysis. Hardware problems occur irrespective of the application running, and links reporting packet squash errors over long time periods are replaced during scheduled maintenance periods. The other types of error messages recorded in the *netwatch* log are *deadlock timeout* and *buffer overruns.* For our preliminary analysis we only analyze the packet squash error messages.

We found a strong correlation between packet squash errors and specific applications, as shown in Table 3. This table shows the range of nodes reporting errors and range of number of error messages whenever certain three applications were running. From prior knowledge of these applications, App-1 is I/O intensive, App-3 is MPI intensive (interprocess communication), and App-4 uses Global Arrays. We found applications that are I/O intensive and non-MPI intensive to generate the fewest packet squash messages. Large numbers of packet squash errors are reported when an MPI-intensive job is running; an even larger number of errors are reported when a job utilizes Global Arrays with MPI. Our interpretations are based on observing multiple runs of the same user application at different times, even though these runs occured along with different mixtures of other running applications during the four month observation period. The results match with the properties of the applications; the interesting finding is that we were able to deduce such application behavior from the*netwatch* error log. Having established a correlation between packet squashes and and link utilization, we are working towards quantifying the observed packet squash error rate to the actual network utilization by an applications in GB/s. With knowledge of such application characteristics we will be able to make better scheduling decisions.

Table 3: Netwatch Log Stats

Applications	App-1	App-3	App-4
# of Compute Nodes	2k	3-5k	5k
% of Nodes Reporting error	0-0.7%	8-10%	15-18%
Error Rate (msg/min)	<2	8-12	15-19

3.2 I/O Usage

Understanding I/O demands of applications is critical for provisioning storage system resources. A specific scientific application can define a generic I/O demand, however the actual I/O utilization is specific to the user running the application. In general, observed patterns of I/O behavior are read at the beginning of a run and write at the end of a run, and for long running jobs, checkpointing at intermediate points. The frequency of checkpointing usually drives the I/O demand of an application. The utilization will also vary depending on how the user performs parallel I/O, e.g., whether or not files are shared between processes. Projecting individual users peak bandwidth (or I/O operations per second) and frequency of file system usage will help provision system resources more efficiently.

In Figure 2, we present a typical usage of the *Spider* file system in a day. The figure has four subplots with each presenting the write bandwidth usage observed during a period of six hours. The write bandwidth usage is sampled in two seconds intervals

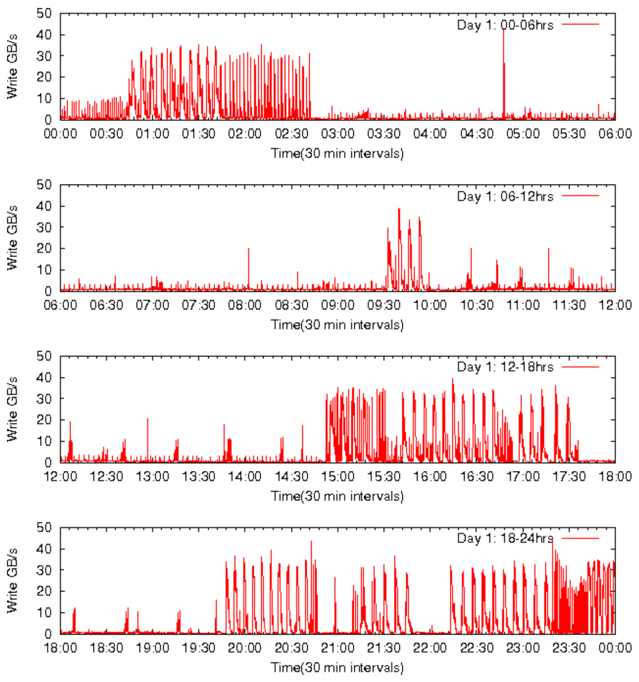

Figure 2: Plot of file system usage observed on a given day.

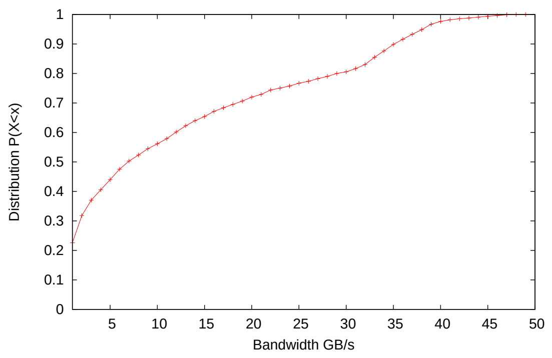

Figure 3: CDF of write bandwidth usage derived from multiple runs of the application (App-1).

from every RAID controller using the manufacturer provided API, and the aggregate value across all controllers gives the total file system usage. From the scheduler's log, our application (App-1) of interest was running during the following time periods: 0:25 to 2:42 hrs, 9:30 to 9:56 hrs, 14:53 to 17:31 hrs, and 19:45 to 23:18 hrs. Observing the write bandwidth usage during the above mentioned time periods we see a clear pattern of high write bandwidth by App-1. Though there might be other applications running concurrently and taxing the file system resources, it is possible to get an estimate of the I/O usage of the application of interest by observing multiple runs of the same application. The bandwidth metrics captured at the controllers are not per application, as that would require extensive application trace information, adding considerable overhead. It is worth mentioning that, we observed an ongoing constant 5 GB/s write activity on backend disks at all times, which is taken into consideration into our framework as background noise.

Using the time series data, we plot the Cumulative Distribution Function (CDF) of the write bandwidth usage by the application (App-1), as shown in Figure 3. From the plot we can infer that for more than 20% of total application runtime the user is writing at a rate greater than 32 GB/s with peaks around 42 GB/s. The CDF plot provides us with an estimate of the user application's I/O needs. In our study of a few other applications, a similar pattern was observable with two other applications, as shown in Table 4. This table summarizes the I/O usage for three applications, with the peak write bandwidth observed for each application and what percentage of the total runtime does the application operate at more than 80% and 50% of the peak write bandwidth. The application (App-1) is a short duration routine that generally runs for a few tens of minutes. However, for a longer running jobs, say 3 hours or more, it is difficult to capture such I/O behavior by directly observing the file system usage. In general for long running applications, users do frequent checkpointing, which is one of the most I/O consuming task. An auto correlation over the runtime stats of the application will give us the periodic I/O usage pattern or the checkpointing pattern of the application. This is evident in the Figure 2, for the time period of 10:00 to 14:30 hrs, as two separate activities with two distinct frequencies with two different amplitudes can be observed.

Table 4: Applications I/O Usage

Applications	App-1	App-2	App-3
Observed Peak(GB/s)	38 -42	12-15	22-35
Runtime >80% of peak	18-20%	6-5%	4-5%
Runtime >50% of peak	38-42%	12-16%	20-25%

4 Related Work

Conventionally, log messages have been associated with temporal data mining techniques, where the frequency and sequence of events are of interest. Recent studies on HPC logs have focused on machine learning and statistical methods for analyzing and detecting system failures. In the machine-learning paradigm described in [15], the log message structures are parsed from the source code and a feature vector is constructed as a sequence of log messages. Using principal component analysis technique, deviations of the run-time log from the predefined vectors are identified, and any variance is defined as an anomaly. SLCT (Simple Logfile Clustering tool [12]) is a data-clustering paradigm for mining event patterns in log data. An *apriori* algorithm, the first stage is to generate a count of all unique words in the log, identify log messages containing words above a threshold value, and then clustering those log lines. This method is based on the assumption that events of interests occur in bursts and ignores errors with low frequency.

Nodeinfo [4], an entropy based anomaly detection system, tags every log message to quantify the importance. Then, the entropy of every node in the system is quantified based on the number of occurrences of alerts within a given time period. It is presumed that all nodes operate similarly, and the entropy of every node should be the same. Any variance of entropy would be categorized as an alert/anomaly. Similarly, the models proposed in [5] and [3] group log messages and use predictive techniques under the assumption that the logs carry all event information and occur in bursts. Recent work [9] [8] have suggested using system logs to understand component level interaction in large scale systems and model the system state leveraging machine learning techniques. First log events or anomalies are correlated to specific hardware, and then the successive events are mapped to other components in the system that are affected by the specific event. This helps understand how system components are interdependent and the cascading effect of system events.

In reviewed papers, one of the principal assumptions in analyzing systems logs is that the system supports logging of all events, which may be impractical for large systems like Jaguar. In peta-scale system, logs in general capture only failure events, which in itself generates a few gigabytes of data per day. In general, the above reviewed papers leverage on machine learning techniques for finding trends or abnormalities that occur multiple times over a period of time. However, it is of interest in identifying such abnormalities on the first occurrence. Our approach of profiling characterizes normal behavior, which can they be used for capturing abnormal activities. Our approach towards using logs for application characterization comes with a profound understanding of the system architecture and applications. This helps correlating events to errors and understanding the impact of such errors on the system and application's performance.

5 Conclusion

As systems have grown to peta-scale the debug levels of logs have decreased (less verbose), while, the volume of log generated has increased. This poses a challenge in terms of readability and the valuable interpretations that can be made from the logs. Apart from the need to enhance the readability of the logs, our approach is focused towards using structured log data for building and supporting analytical tools that can enhance our interpretations of the log. Currently, we are working on presenting this information via a web interface in a more interactive manner. Also, we are extending our work by building a profiling tool to model the runtime characteristics of an individual application, which can help in anomaly detection, identify inter-job interference, and lead to the design of context-aware schedulers. Profiling an application gives us the expected or acceptable behavior of an application that providing the capability to identify anomalous behavior, which is a deviation from the expected behavior. Also, such deviations can help us identify applications that tend to get impacted or have poor performance because of other applications running on the shared compute platform. Identifying such inter-job interference can help us make well-informed scheduling decisions increasing the overall throughput of the system.

References

[1] Totalview. *http://www.roguewave.com/support/product-documentation/index.html.aspx.*

[2] Using cray performance analysis tools. *Document S-2474-51, Cray User Documents (http://docs.cray.com)*, 2009.

[3] A. Makanju, A.N. Zincir-Heywood, E. E. Milios. Clustering event logs using iterative partitioning. In *ACM SIGKDD International conference on Knowledge discovery and data mining*, 2009.

[4] J. Oliner, A. Aiken, and J. Stearley . Alert Detection in System Logs. In *Proceedings of the International Conference on Data Mining (ICDM)*, 2008.

[5] Y. Liang, Y. Zhang, H. Xiong, Hui, R. Sahoo. Failure Prediction in IBM BlueGene/L Event Logs. In *Proceedings of the International Conference on Data Mining (ICDM)*, 2007.

[6] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marini, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[7] Hwa-Chun Wendy Lin. Understanding aprun use patterns. In *Cray User Group Conference*, 2009.

[8] J. Oliner, A. Aiken. A query language for understanding component interactions in production systems. In *Proceedings of the International Conference on Supercomputing(ICS)*, 2010.

[9] J. Oliner, A. Aiken. Online detection of multi-component interactions in production systems. In *Proceedings of the International Conference on Dependable Systems and Networks(DSN)*, 2011.

[10] R. Gunasekaran, D. Dillow, B. Park, G. Shipman, D. Maxwell, J. Hill, A. Geist. Corelating log mesages for system diagnostics. In *Cray User Group Conference*, 2010.

[11] R. Miller, J. Hill, D. Dillow, R. Gunasekaran, D. Maxwell. Monitoring tools for large scale systems. In *Cray User Group Conference*, 2010.

[12] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. In *IEEE IPOM'03 Proceedings*, 2003.

[13] G. M. Shipman, D. A. Dillow, S. Oral, and F. Wang. The spider center wide file systems; from concept to reality. In *Cray User Group Conference*, 2009.

[14] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting performance data with papi-c. In *3rd Parallel Tools Workshop*, 2009.

[15] W. Xu, L. Huang, A. Fox, D. Patterson, M. Jordan. Mining Console Logs for Large-Scale System Problem Detection. In *3rd Workshop on Tackling System Problems with Machine Learning Techniques(SysML)*, 2008.