

Experience Mining Google’s Production Console Logs

Wei Xu* Ling Huang[†] Armando Fox* David Patterson* Michael Jordan*
*University of California at Berkeley [†]Intel Labs Berkeley
{xuw,fox,pattnsn,jordan}@cs.berkeley.edu ling.huang@intel.com

Abstract

We describe our early experience in applying our console log mining techniques [19, 20] to logs from production Google systems with thousands of nodes. This data set is five orders of magnitude in size and contains almost 20 times as many messages types as the Hadoop data set we used in [19]. It also has many properties that are unique to large scale production deployments (e.g., the system stays on for several months and multiple versions of the software can run concurrently). Our early experience shows that our techniques, including source code based log parsing, state and sequence based feature creation and problem detection, work well on this production data set. We also discuss our experience in using our log parser to assist the log sanitization.

1 Introduction

Free text console logs are generated and collected from complex Internet service systems, even in systems that are extensively monitored and as large as Google’s. Although these logs contain detail and useful information for problem diagnosis, they are rarely used by anyone other than the developer herself. For system operators who are usually assigned to fix specific problems, console logs are often too large to examine manually and too unstructured to analyze automatically.

In our earlier work [19, 20], we proposed a general methodology to explore the intrinsic structures and patterns in console logs, automatically detect performance problems, and often reveal root causes of the problems. Our experiments show that our method works effectively on two real world open source systems, Hadoop and Darkstar. In both cases, we can detect anomalies with high precision/recall from tens of millions lines of log messages, greatly reducing the amount of information an operator needs to manually examine.

Although all case studies presented in [19, 20] are real open source systems, the logs we used came from experimental deployments. In this paper, we discuss our early experience applying our log mining techniques to data from Google’s production systems.

We studied console logs from three systems at Google. We focus on only one system, which we call GX, in this paper. GX is a distributed storage system consisting of thousands of nodes in a Google production cluster. There are several independent monitoring systems built into GX including one for collecting console logs. GX console logs are plain text files on local disks of each node. Our data set contains most of the log messages in a two-month period¹.

We discussed our log analysis techniques in detail in [19, 20], and we summarize these techniques in Section 2. Generally speaking, it is fairly easy to adapt our console log mining techniques to process GX logs, and find interesting information about the system. However, there are several new challenges posed by this data set which we believe is representative in large scale production clusters.

Data size. This data set is five orders of magnitudes larger than the Hadoop data set used in [19]. In contrast to Hadoop, which logs each write/read operation, GX only logs significant events such as errors and periodic background tasks. Even omitting per-operation logs, a single node can generate over half a million messages per day. Overall the system generates over a billion messages per day. Our entire data set contains tens of billions of messages, with a size about 400 GB uncompressed. We have to parallelize the parsing and feature creation steps

¹As the logs are collected on local disks only, there are several corruptions due to disk/node failures or software errors.

onto thousands of nodes to process the entire dataset within a couple of hours.

Many message types / variables. Because of the complexity of GX and its extensive interactions with other infrastructure components, there are hundreds of different message types in the logs. Our source code analysis found over 20,000 different possible message types in GX, compared to only about 1,000 in Hadoop. Most of these types never appear in the log because of the logging level settings. There is a wide variety of message variables (both identifiers and state variables) as well. Though we could apply our methods on each identifier and each state variable, it would cost more computationally than we could afford. Thus, we heuristically choose the message variables that are likely to indicate problems.

Longer time period with multiple versions. Unlike the data sets we analyzed in [19], which lasts only two days, we have almost two months worth of data. Due to the longer time period, we need to use the sequence segmentation techniques, even for an offline analysis.

Two limitations prevent us from quantitatively evaluating detection results. First, console logs in GX are not regularly used by system operators, and thus there are no manual labels explaining the logs. With an unclear “ground truth”, it is not possible to evaluate metrics such as true/false positives. Second, due to confidentiality issues, all human readable string constants are removed (details in Section 4), rendering it impossible to understand the semantics of patterns and anomalies discovered. Due to these limitations, we only evaluate our results qualitatively and show that they are still useful to the developers and operators of system GX.

Our major contribution in this paper is the experimental evaluation of our log mining techniques proposed in [19, 20] on large scale data sets from production systems, which further demonstrates the generality and applicability of these techniques. We also introduce techniques for removing sensitive information from console logs, a minor yet essential step in many research projects involving real world log data.

In the rest of the paper, we summarize our techniques introduced in [19, 20] in Section 2, discuss the implementation and effectiveness of our message parser in Section 3, and briefly describe the log sanitization with a focus on how the message parser helps in this process in Section 4. We discuss both state ratio vector based and message count vector based problem detections in

Section 5, review existing work in this area in Section 6, and finally discuss about future directions and conclude in Section 7.

2 Review of Our Techniques

Our technique involves three major steps, log parsing, feature creation and machine learning. Each step not only reduces the amount of data to process, but also transforms the free text logs into more structured and less noisy data. We summarize the general ideas of our log mining techniques in this section. Readers may refer to [19, 20] for details.

Step 1: Log Parsing. The log parsing step explores the structure within a single message. During this step, we extract *message types* and *message variables* from a free text message. Message types are marked by constant strings in a log message. For example, in the log message `starting: xact 346 is ABORTING`, a human operator may easily guess that `starting: xact` and `is` are string constants, which indicate the message type, while `346` and `ABORTING` are message variables. By converting a log message to message types and variables, we eliminate most of the complexity of dealing with the free texts in further steps. As a beneficial side effect, the log parsing step also eliminates corrupted messages, which is common in GX logs, due to the scale of GX.

Existing log parsing tools [17, 18, 12, 4, 6] use various data mining methods to discover the message structures from historical logs. Different from these methods, we use static source code analysis to extract all possible log printing statements, from which we generate message templates (i.e. regular expressions). We also infer the data types of all message variables. This process of source code analysis is programming language specific. In addition to the Java parser used in [19], we implemented a new source analyzer for C/C++, in order to handle GX source code.

After extracting message templates, we build an index on all possible message templates. During runtime, the log parsing is simply a search through the index for the best-matching template. Note that the runtime log parsing is stateless once the index is built and distributed, so it is easy to parallelize the process. In fact, in order to process tens of billions of lines of logs from GX, we distribute the parsing job to thousands of nodes as a map-reduce style job, which enables us to process the entire data set in a couple of hours.

Step 2: Feature Creation. Message parsing only exploits the log structures within a single message. However, many problems can only be detected when examining a sequence of related messages. In [19], we intro-

duced two different ways to group messages, leading to two kinds of problem detection. Both proved to be useful in the GX case. We briefly summarize the mining techniques we used, and readers may refer to [19] for more details.

The first feature we created is called *state ratio vector*. This feature is based on *state variables*, the labels enumerating a set of possible states an object could have in program. For example, HTTP return code (200, 404, etc) and transaction termination state (OK, Abort, etc) are examples of state variables. Construction of the state ratio vectors is simple: each dimension of the vector represent a state, and the value of the dimension is how many messages reports that state. Intuitively, the feature is a snapshot of the *global* state of the system at certain time period.

The other feature, *message count vector*, captures an execution trace for an *individual* session. It is based on *identifiers*, which are message variables used to identify objects manipulated by the program (e.g., the transaction ids, file names, etc). To form the message count vector, we first automatically discover identifiers, then group together messages with the same identifier values, and create a vector per group. Each dimension of the vector corresponds to a different message type. The value of the dimension is the number of messages of the corresponding type in the message group. The feature is similar to an execution trace [5].

Both features were useful in GX logs. We further discuss their applications to GX in Section 5.

Step 3: Machine learning. Although state ratio vectors and message count vectors are semantically different, they are similar in their numerical form. We find that the Principal Component Analysis (PCA)-based anomaly detection method [3] works well for both features.

We used a PCA-based method to discover the statistically dominant patterns and thereby identify anomalies inside data. Given high-dimensional data, PCA computes a (small) set of coordinates—the *principal components*—that reflect dominant covariation among the original coordinates. PCA detector captures the dominant pattern in a transformation matrix \mathbf{PP}^T , where \mathbf{P} is formed by the top principal components estimated from training data (archived and/or periodically updated). In the detection phase, the *abnormal component* of the incoming data vector \mathbf{y} is computed as $\mathbf{y}_a = (\mathbf{I} - \mathbf{PP}^T)\mathbf{y}$, i.e., \mathbf{y}_a is the projection of \mathbf{y} onto the abnormal subspace. The *squared prediction error* $\mathbf{SPE} \equiv \|\mathbf{y}_a\|^2$ (squared length of vector \mathbf{y}_a) is used for detecting ab-

normal events: We mark vector \mathbf{y} as abnormal if

$$\mathbf{SPE} = \|\mathbf{y}_a\|^2 > Q_\alpha, \quad (1)$$

where Q_α denotes the threshold statistic for the **SPE** residual function at the $(1 - \alpha)$ confidence level, which can be automatically determined [10].

Two-Stage Detection. Message count vectors, however, assume that we have access to all messages about a single identifier at once. This is unrealistic especially in a system that stays up for months. We used frequent pattern mining techniques to segment the long-lasting sequence into *subsequences*, or *sessions*, and construct message count vectors from these sessions. A session contains logically related messages we want to analyze as a group, while lasts only a short time. In [20], we describe the approach as a *two-stage detection technique*, where we detect “normal” sessions using frequent pattern based methods, while handling the less common sessions with PCA based anomaly detection.

Although this two-stage detection technique is designed to perform timely online anomaly detection in [20], it is useful in the GX case even in a batch setting, because the logs contain two-month worth of data and many logically independent sessions could refer to the same identifier (e.g. the data can be read many times during independent sessions).

3 Log Parsing

Automated log parsing is essential in analyzing GX logs. GX, like almost all other Google systems, is continuously updated, despite being a relatively mature production system.

Figure 1 plots the number of new log printing statements added to the source code each month in four Google systems during the past six years². Systems 1, 2, 3 are relatively mature systems, while System 4 is a new development. We can see that there are tens or even hundreds of new log printing statements introduced every month independent of the development stage. These statements are added either to test new functionalities or to debug pre-existing problems. It is very difficult to maintain manually-written rules with such a high churn rate, and an automated message parsing solution is unavoidable.

Most console log messages at Google have a special field, the source code line number, which is generated by the logging library. One initial thought was that we could use the line number field as the message type, eliminating the need of log parsing. However, the rate of code

²GX is one of them, real system names are hidden.

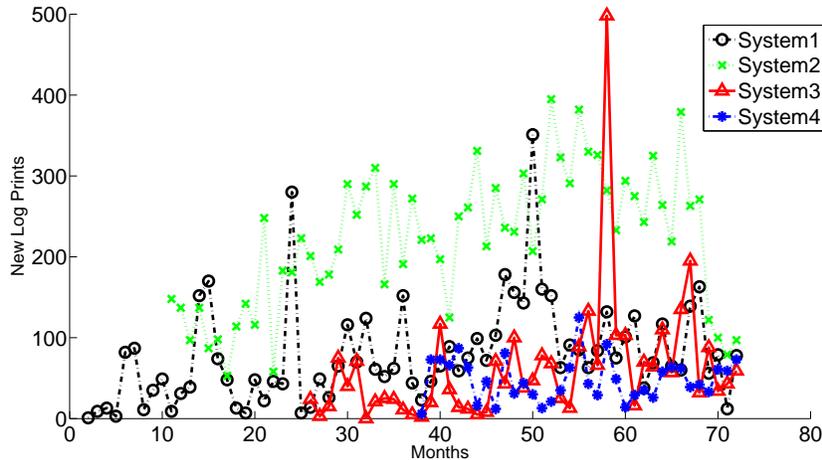


Figure 1: Number of new log printing statements added each month in four real Google systems. System 1, 2, 3 are relatively mature systems, while System 4 is new development. We can see that many new message types are introduced at all stages of development.

churn renders this field less useful. A single code change might affect the line numbers of many different message types, though the semantics of such message types stay the same. Even worse, there can be multiple versions of GX running in the same cluster, due to the software rollout process. Patching the line numbers across different versions is a painful process. On the other hand, our log parser can easily avoid this concurrent version problem. Using the string constants embedded in the messages to identify message types, the log parser can easily tell which message types stay the same across versions and which are new or modified types.

GX is written in C and C++. The parser we described in [19] handles Java only. We developed a new C/C++ parser for the data set. We briefly describe its implementation and parsing accuracy in the remaining of this section.

3.1 C/C++ Parser Implementation

C++ programmers use both *printf* style format strings and C++ stream (`cout`) style string concatenations. As we discussed in [19], it is conceptually simple to extract message types if *printf* style formatting strings are used. C++ streams are similar construct as the string concatenations in Java. In practice, extracting templates from C/C++ programs is complicated by the heavy use of pre-processor macros.

We use the built-in parser for Eclipse C Development Tooling (CDT) [7] to parse both C and C++ code into

an AST. We use an implementation similar to our Java parser to traverse the AST to extract all log printing functions and their parameters.

C/C++ programmers heavily utilize macros, which complicate our analysis. Macros are handled by the pre-processor before compiling, and thus are not part of the AST. To make things worse, macro expansion is determined by external information, such as the pre-processor arguments passed to the compiler at build-time and header files that are OS-dependent.

We cannot simply ignore these macros, as programmers use macros extensively for logging; ignoring these macros prevents the resulting program from compiling correctly. Our analyzer could analyze the build descriptions (e.g. `makefile`) to understand what command line flags are used. However, we face the cross-compiling issue: unless the system we use for static analysis is exactly the same as the system generating the logs, the templates might still be different due to the system-dependent macros.

Instead, our analyzer works to evaluate the macro with all possible branches, and takes the union of all the message templates extracted from each branch. In this way, we produce more possible templates than actually exist, but we care more about completeness than the extra templates. These templates use a small amount of extra space in the index, but do not affect our log parsing accuracy as they never match any log messages during the parsing stage.

Google’s systems have complex inter-dependencies of libraries and packages. GX depends on many other projects, which may print their own logs. It is a challenge to get all source files that could print logs. We currently use heuristics to find commonly used projects, and use full text search on the entire source repository for log messages that do not match any template to find possible source files.

Although C++ is an object-oriented language, GX does not heavily use the object-oriented features for log printing. Thus, we did not perform type resolution steps as we did for Java. Instead, we recognize data type of variables using type declarations and `printf` format strings.

3.2 Log Parsing Evaluation

System	Total Log Lines	Failed %
Google System 1	46×10^9	<0.001%
Google System 2	8×10^9	<0.012%
Google System 3	8×10^9	<0.011%

Table 1: Parsing Accuracy on Google log data. GX log is a subset of a dataset listed in the table.

Table 1 summarizes the message parsing accuracy on logs from three different Google production systems. We parsed source code for all libraries referenced by each of the system in order to cover as many message types as possible.

While accurately determining the log parsing accuracy is not possible, we estimate the error rate using a heuristic-based checker. The checker, built as part of the sanitization program described in the next section, checks each parsing result to ensure the message type and message variables are extracted. The checker also prints unique message types and parser failure cases for human verification.

The “Failed %” field in Table 1 is calculated from the output of this checker. Failure cases mostly involve complex array dumps (e.g. programmer could use a function to dump an entire array in a single log message), and long string variables, such as a command line with tens of arguments. As these corner cases are relatively rare in the data, we can still achieve high parsing accuracy. This high accuracy is essential for us to perform further steps, especially the sequence based detection.

4 The Art of Log Sanitization

As a usual requirement for using real world production data, we have to remove sensitive information before mining them. We call the process *log sanitization*.

Though sanitizing console logs is not a major goal of this research, it is a necessary step to obtain the data for our research. Our log parsing technique made it easier to accomplish at low human cost. We could not evaluate the effectiveness of sanitizing such logs, lacking of a clear standard for “clean” logs, but our technique is effective enough so we were allowed to use the data in our research. In this section, we highlight some experience in the log sanitization process.

There are potentially two separate types of sensitive information in console logs: 1) program logic and 2) sensitive data, such as internal entity names. Due to their free text nature, console logs may reveal this sensitive information in any message. Hashing each English word does not work because many identifiers are hierarchical (e.g. a path or a URL), and the naive hashing would destroy these hierarchies and make the identifier-based grouping impossible.

Our message parsing makes the sanitization easier and more effective. Figure 2 illustrates major steps to sanitize a message. The message parser first identifies string constants in the message, and then replaces these constants with a unique message type ID. This step removes semantic meanings of messages, making it hard for adversaries to guess the internal program logic. Note that this step is reversible as long as the message templates are available, making it possible for Google’s internal operators to reconstruct the original message.

The next step is to apply a one-way secure hash function to all message variables (or parts of the variables) containing sensitive information. In this step, our goal is to preserve as much information as possible while removing all sensitive data.

Selecting which variables to hash can be difficult. One simple scheme is to obtain a list of “sensitive terms”, look for variables containing these terms and hash them. This scheme does not work well enough because 1) it is not possible to obtain an exhaustive list of all sensitive terms, and 2) some of these sensitive terms are phrases instead of single words. Arbitrary string concatenations, especially when constructing hierarchical names, make it hard to identify all phrases from a long string variable. Solving the general problem of string segmentation is a hard problem in information retrieval [11, 21]. Fortunately, we have data type information for each variable from the message parser, so we can specify sanitization rules for each *data type*, rather than each individual variable. For example, if we know many variables are of URL type, we only need to specify a rule to handle URLs, rather than specifying each variable. Although GX contains almost 2000 different variables,



Figure 2: Log sanitization overview. We need to sanitize two separate types of sensitive information: 1) program logic, and 2) sensitive data. Our sanitization is based on structures of log messages as well as a list of sensitive words.

there are only about 20 different data types. Using data-type-based rules significantly reduces the manual work in the sanitization process.

There are exceptions to the data-type-based rules. For example, we want to keep some integer-typed variables (e.g. performance counters) while removing other integers revealing machine configurations. We developed a scripting engine that allows us to specify these exceptions. It took us only a few days to test and fix these exceptional cases, which is far more efficient than manually examine all different message types.

We applied the sanitization process discussed above to GX logs. To informally evaluate the effectiveness of such methods, we generated a random sample of 100,000 messages. The random sample passed reviews by Google operators. However, the review is subjective due to the lack of a standard of a “clean” log, and thus we were not able to share the data with the research community. We believe this problem is common with many research involving real production data.

5 Problem Detections

We apply both state based and sequence based detections on GX data, each of which yields certain interesting patterns and problems. Due to the lack of “ground truth”, we are not able to quantitatively evaluate these results. However, the results are intuitive and insightful for the system operators.

5.1 Global State Based Detection

Most important anomalies are those related to problems affecting a large part or the entire system. The *state ratio vector* feature in [19], capturing the global state at a certain time window, is especially suitable in detecting and diagnosing such anomalies.

We used the message type ID obtained from the parsing step as a special state variable. Most of the messages in GX logs are generated by background tasks reorganizing the data storage. The performance and correctness impact of such tasks can be significant. Thus, which

background task is running at a given time is an important system state to capture. As each background task always generates a fixed set of message types that are distinct from types generated by other tasks, capturing message type IDs is equivalent to capturing the current state of running background tasks.

We count the number of each different message type occurring in the entire system in every 10-minute time window. There are over 400 different message types, so the resulting state ratio vectors have over 400 dimensions.

We then apply PCA detection to these vectors, and plot in Figure 3 the Squared Prediction Error (SPE, defined in Equation (1)) for each time window. We see that the state ratio vectors stay normal during most of the time windows, indicating that the relative rate for each message type remains constant, matching our assumption that the relative frequency of different background tasks remains constant.

There are a few anomalous time windows, shown as spikes in Figure 3. Because we do not have access to semantic information of each message type nor do we understand the internals of GX, it is not possible to interpret these anomalies. Instead, to verify that these anomalies did affect system performance, we checked performance alarm data. There is a monitoring system that periodically probes GX and raises alarms to operators if there is significant performance degradation. The operators then investigate these alarms and attach their comments. The text comments with arrows in Figure 3 show these alarms with comments. Each arrow points to the time when the alarms are generated. We can easily see that these alarms happen at almost the same time as the anomalies detected by PCA, suggesting changes in state ratio vector feature are correlated with performance problems. Note that there is a complex logic in the alarm system to suppress unnecessary/duplicate alarms in order to reduce the operator’s manual tasks. Due to these rules, the alarms and the anomalies we detected are not likely to form a one-to-one correspondence. Thus, these alarms are different

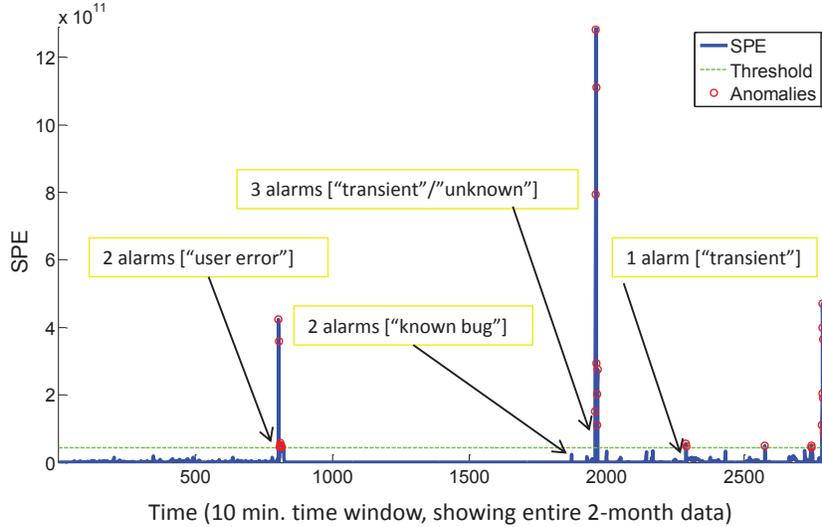


Figure 3: GX problem detection results using message type counts on the entire two-month period. SPE (y-axis) is calculated using Equation (1). Texts with arrows are automatically generated alarms as well as operators’ comments on these alarms. The arrows point to the time when the alarms happen.

from the manual labels we had in the Hadoop and Darkstar cases in [19], preventing us from accurately evaluating the false positive/negative rates.

With the alarm system built-in, detecting anomalies becomes a less important goal for our log analysis. However, it is not easy for the operators to determine the cause of the performance problems just by looking at the alarms, as the alarms provide no information other than a long latency. In Figure 3, half of the alarms are not clearly marked with the actual cause. On the other hand, abnormal log patterns output by our method contain information about system internal states, and thus can be useful supplements for operators to better diagnose these performance anomalies.

5.2 Sequence-based Detection

State variable based detection helps explain the global performance crisis. On the other hand, system developers are also interested in finding anomalies associated with individual background tasks. These anomalies might not cause a global crisis, but affect the performance in certain parts of the system or even indicate underlying bugs. We used the message count vector method to analyze sequence-based features.

In GX, storage is partitioned onto multiple nodes. Each partition goes through a similar life cycle: it is first created, then migrated among multiple nodes, and finally terminated by a re-partition or deletion operation. A par-

tion is manipulated by background tasks, which generate sequences of log messages. In fact, the majority of console log messages are generated by such background tasks.

Each partition is identified by a partition identifier, which is a complex string of arbitrary length and format. Grouping messages by such identifiers, we can capture all messages about events happening on a partition. Each message sequence can last a long time, some even spanning the entire two-month period. In this case, the approach in [19], which builds message count vectors from the entire sequence, becomes less useful. In order to segment each sequence into sessions, we adopted the two-stage detection method described in [20].

In the two-stage approach, we first applied frequent pattern based methods [20] to segment the sequences into sessions. This pattern mining step outputs a set of intuitive patterns. For example, when a partition is migrated to a node, the original node normally prints five messages for unloading the partition, while the receiving node prints six messages for loading the node. Our pattern mining technique is able to accurately capture these frequently appearing sessions, as well as the time distribution for each session to complete. We discovered nine different patterns, each of which contains two to six events and represents a common background task in the system. Due to confidentiality issues, we omit the details about the patterns here. There are some less fre-

quent tasks, such as repartitioning, not captured as frequent patterns. Although we could have lowered the minimal support requirement during the pattern mining, we decide to leave these less frequent patterns to the PCA method in the second stage.

In the second stage PCA detection marked less than 0.1% of over 200 million sequences as anomalies. Most of the anomalous sessions either have rare message types (probably error messages), or take an abnormally long time to complete the session. Without a deep understanding of GX internals, we were not able to determine the performance/correctness impact of each anomaly, nor could we estimate the false positive rate. However, informal communications with GX operators show that these anomalies could help GX developers to better understand corner cases and further improve the performance and reliability of the system.

6 Related Work

Traditionally, operators use *grep* or write ad-hoc scripts (mostly in Perl or Python) for log analysis. To scale the approach to large log data, researchers have proposed to parallelize these scripts (e.g., Google’s Sawzle [8]) or adopt a full text search based approach such as Splunk [15]. These approaches require the operators to understand the semantics of log messages and internals of the system, which is not always possible. Other popular improvement to these ad-hoc scripts is rule-based systems. The most popular ones include Logsurfer [14] and Swatch [9]. However, rules are extremely difficult to maintain in a system that is continuously upgraded as GX.

[16] uses textual properties to capture the global message patterns during certain time window. It is similar to our state-ratio based detection, but certain English words can be overloaded in different message types, making the process less reliable than our approach based on message types. In fact, our approach is more similar to sequence based detections, traditionally done with structured traces [2, 1, 5].

There are methods that could detect problems with a single message [22]. However, the computational complexity prevents it from scaling to large data sets. [13] analyzed *syslog* from a set of super computers. The size of the data set is comparable to ours, but the nature of the data are different: ours are mainly debugging information from a single distributed system, while the logs described in [13] are mostly OS-level alarms.

7 Conclusion and Future work

In this paper, we summarized our early experience of applying the console log mining methodology to a produc-

tion system at Google. Despite the size and complexity of the data set, we show that the parsing, feature creation, pattern mining, and anomaly detection techniques are easy to adapt to system GX and yield promising results.

Lacking of even basic ground-truth about GX events prevents us from formally evaluating our results, but informal communications with operators show that the results are insightful. State based detections help find root causes of performance problems, while sequence based detection is useful for finding hard-to-notice corner error cases, helping developers further improve the system.

This project is a work-in-progress, and there are many potential future directions to pursue. First, many problems are caused by the resource contention among infrastructures and applications that share the same cluster platform. Analyzing logs from different applications concurrently may reveal these hard-to-detect resource contentions, and help prevent such problems. Second, other than console logs, there are many different kinds of monitoring information at different granularity on each cluster node. Combining them with console logs may help reveal and diagnose more subtle problems. Last but not least, console logs on large clusters may take nontrivial amount of resource to generate and maintain. However, there is currently no quantitative benchmark on how much resource is used for logging and monitoring. Creating such benchmark can be essential for designing future log/monitoring system.

8 Acknowledgments

The authors thank Google for providing access to the production log data and many Googlers offered help on every stage in the project. Special thanks to Urs Hoelzle, Alex Wu, Jay Sutaria for hosting this project, Lea Kissner, Ante Derek, Breno de Medeiros for the advices on log sanitization, and Shoumen Saha for explaining tech details of the GX system. Urs Hoelzle and Ariel Rabkin reviewed earlier version of this paper provided invaluable suggestions.

This research is supported in part by gifts from Sun Microsystems, Google, Microsoft, Amazon Web Services, Cisco Systems, Cloudera, eBay, Facebook, Fujitsu, Hewlett-Packard, Intel, Network Appliance, SAP, VMWare and Yahoo! and by matching funds from the State of California’s MICRO program (grants 06-152, 07-010, 06-148, 07-012, 06-146, 07-009, 06-147, 07-013, 06-149, 06-150, and 07-008), the National Science Foundation (grant #CNS-0509559), and the University of California Industry/University Cooperative Research Program (UC Discovery) grant COM07-10240.

References

- [1] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, San Francisco, CA, 2004.
- [2] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN'02)*, Washington, DC, 2002.
- [3] R. Dunia and S. J. Qin. Multi-dimensional fault diagnosis using a subspace approach. In *Proc. of American Control Conference (ACC'97)*, Albuquerque, NM, 1997.
- [4] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: fully automatic tool generation from ad hoc data. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, San Francisco, CA, 2008.
- [5] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. Xtrace: A pervasive network tracing framework. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI'07)*, Cambridge, MA, 2007.
- [6] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining (ICDM'09)*, Washington, DC, 2009.
- [7] E. Graf, G. Zraggen, and P. Sommerlad. Refactoring support for the C++ development tooling. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, Montreal, Canada, 2007.
- [8] R. Griesemer. Parallelism by design: data analysis with sawzall. In *CGO '08: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, Boston, MA, USA, 2008.
- [9] S. E. Hansen and E. T. Atkins. Automated system monitoring and notification with Swatch. In *LISA '93: Proceedings of the 7th USENIX conference on System administration*, Monterey, CA, 1993.
- [10] J. E. Jackson and G. S. Mudholkar. Control procedures for residuals associated with principal component analysis. *Technometrics*, 21(3), 1979.
- [11] C. Kit, H. Pan, and H. Chen. Learning case-based knowledge for disambiguating chinese word segmentation: a preliminary study. In *Proceedings of the first SIGHAN workshop on Chinese language processing*, Morristown, NJ, 2002. Association for Computational Linguistics.
- [12] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios. Clustering event logs using iterative partitioning. In *KDD '09: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, Paris, France, 2009.
- [13] A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *Proceedings of the 2007 International Conference on Dependable Systems and Networks (DSN'07)*, Edinburgh, UK, 2007.
- [14] J. E. Prewett. Analyzing cluster log files using logsurfer. In *Proceedings of Annual Conference on Linux Clusters*, 2003.
- [15] Splunk Inc. Splunk user guide. <http://www.splunk.com/>, Sept 2008.
- [16] J. Stearley. Towards informatic analysis of syslogs. In *LUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, Washington, DC, 2004.
- [17] R. Vaarandi. A data clustering algorithm for mining patterns from event logs. *Proceedings of the 2003 IEEE Workshop on IP Operations and Management (IPOM'03)*, 2003.
- [18] R. Vaarandi. A breadth-first algorithm for mining frequent patterns from event logs. In *INTELLCOMM*, volume 3283. Springer, 2004.
- [19] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Large-scale system problems detection by mining console logs. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, Big Sky, MT, 10 2009.
- [20] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Online system problem detection by mining patterns of console logs. In *ICDM '09: Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, Miami, FL, 12 2009.
- [21] Y. Yao and K. T. Lua. Splitting-merging model of chinese word tokenization and segmentation. *Nat. Lang. Eng.*, 4(4), 1998.
- [22] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pappas. Sherlog: error diagnosis by connecting clues from run-time logs. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, Pittsburgh, PA, 2010.