DOI:10.1145/2076450.2076466

Article development led by acmqueue
queue.acm.org

Logs contain a wealth of information to help manage systems.

BY ADAM OLINER, ARCHANA GANAPATHI, AND WEI XU

Advances and Challenges in Log Analysis

COMPUTER-SYSTEM LOGS provide a glimpse into the states of a running system. Instrumentation occasionally generates short messages that are collected in a system-specific log. The content and format of logs can vary widely from one system to another and even among components within a system. A printer driver might generate messages indicating that it had trouble communicating with the printer, while a Web server might record which pages were requested and when.

As the content of the logs is varied, so are their uses. The printer log might be used for troubleshooting, while the Web-server log is used to study traffic patterns to maximize advertising revenue. Indeed, a single log may be used for multiple purposes: information about the traffic along different network paths, called *flows*, might help a user optimize network performance or detect a malicious intrusion; or call-detail records can monitor who called whom and when, and upon further analysis can reveal call volume and drop rates within entire cities.

This article provides an overview of some of the most common applications of log analysis, describes some of the logs that might be analyzed and the methods of analyzing them, and elucidates some of the lingering challenges. Log analysis is a rich field of research; while it is not our goal to provide a literature survey, we do intend to provide a clear understanding of why log analysis is both vital and difficult.

Many logs are intended to facilitate debugging. As Brian Kernighan wrote in *Unix for Beginners* in 1979, "The most effective debugging tool is still careful thought, coupled with judiciously placed print statements." Although today's programs are orders of magnitude more complex than those of 30 years ago, many people still use `printf` to log to console or local disk, and use some combination of manual inspection and regular expressions to locate specific messages or patterns.

The simplest and most common use for a debug log is to `grep` for a specific message. If a server operator believes that a program crashed because of a network failure, then he or she might try to find a "connection dropped" message in the server logs. In many cases, it is difficult to figure out what to search for, as there is no well-defined mapping between log messages and observed symptoms. When a Web service suddenly becomes slow, the operator is unlikely to see an obvious error message saying, "ERROR: The service latency increased by 10% because bug X, on line Y, was triggered." Instead, users often perform a search for severity keywords such as "error" or "failure". Such severity levels are often used inaccurately, however, because a developer rarely has complete knowledge of how the code will ultimately be used.

Furthermore, red-herring messages (for example, "no error detected") may pollute the result set with irrelevant events. Consider the following message from the BlueGene/L supercomputer:

```
YY-MM-DD-HH:MM:SS NULL RAS
BGLMASTER FAILURE ciodb exit-
ed normally with exit code 0
```

The `FAILURE` severity word is unhelpful, as this message may be generated during nonfailure scenarios such as system maintenance.

When a developer writes the `print` statement of a log message, it is tied to the context of the program source code. The content of the message, however, often excludes this context. Without knowledge of the code surrounding the `print` statement or what led the program onto that execution path, some of the semantics of the message may be lost—that is, in the absence of context, log messages can be difficult to understand.

An additional challenge is that log files are typically designed to represent a single stream of events. Messages from multiple sources, however, may be interleaved both at runtime (from multiple threads or processes) and statically (from different modules of a program). For runtime interleaving, a thread ID does not solve the problem because a thread can be reused for independent tasks. There have been efforts to include message contexts automatically (X-Trace,[4] Dapper[12]) or to infer them from message contents,[15] but these cannot completely capture the intents and expectations of the developer.

Log analysis can help optimize or debug system performance. Understanding a system's performance is often related to understanding how the resources in that system are used.

The static interleaving scenario is more challenging because different modules may be written by different developers. Thus, a single log message may have multiple interpretations. For example, a "connection lost" message might be of great importance to the author of the system networking library, but less so for an application author who is shielded from the error by underlying abstractions. It is often impossible for a shared-library author to predict which messages will be useful to users.

Logging usually implies some internal synchronization. This can complicate the debugging of multithreaded systems by changing the thread-interleaving pattern and obscuring the problem. (This is an example of a so-called heisenbug.) A key observation is that a program behaves nondeterministically only at certain execution points, such as clock interrupts and I/O. By logging all the nondeterministic execution points, you can faithfully replay the entire program.[7,14] Replay is powerful because you can observe anything in the program by modifying the instrumentation prior to a replay. For concurrent programs or those where deterministic execution depends on large amounts of data, however, this approach may be impractical.

Log volume can be excessive in a large system. For example, logging every acquire and release operation on a lock object in order to debug lock contention may be prohibitively expensive. This difficulty is exacerbated in multimodule systems, where logs are also heterogeneous and therefore even less amenable to straightforward analysis. There is an inherent cost to collecting, storing, sorting, or indexing a large quantity of log messages, many of which might never be used. The return on investment for debug logging arises from its diagnostic power, which is difficult to measure.

Some users need aggregated or statistical information and not individual messages. In such cases, they can log only aggregated data or an approximation of aggregated data and still get a good estimate of the required statistics. Approximation provides statistically sound estimates of metrics that are useful to machine-learning analyses such as PCA (principal component analysis) and SVM (support vector machine[8]). These techniques are critical in networked or large-scale distributed systems, where collecting even a single number from each component carries a heavy performance cost. This illustrates the potential benefits of tailoring instrumentation to particular analyses.

Machine-learning techniques, especially anomaly detection, are commonly used to discover interesting log messages. Machine-learning tools usually require input data as numerical *feature vectors*. It is nontrivial to convert free-text log messages into meaningful features. Recent work analyzed source code to extract semi-structured data automatically from legacy text logs and applied anomaly detection on features extracted from logs.[15] On several open source systems and two Google production systems, the authors were able to analyze billions of lines of logs, accurately detect anomalies often overlooked by human eyes, and visualize the results in a single-page decision-tree diagram.

Challenges remain in statistical anomaly detection. Even if some messages are abnormal in a statistical sense, there may be no further evidence on whether these messages are the cause, the symptom, or simply innocuous. Also, statistical methods rely heavily on log quality, especially whether "important" events are logged. The methods themselves do not define what could be "important."

Static program analysis can help discover the root cause of a specific message by analyzing paths in the program that could lead to the message. Static analysis can also reveal ways to improve log quality by finding divergence points, from which program execution might enter an error path; such points are excellent candidates for logging instrumentation.[16] Static analysis techniques are usually limited by the size and complexity of the target system. It takes hours to analyze a relatively simple program such as Apache Web Server. Heuristics and domain knowledge of the target system usually make such analyses more effective.

Performance

Log analysis can help optimize or debug system performance. Understanding a system's performance is often related to understanding how the resources in that system are used. Some logs are the same as in the case of debugging, such as logging lock operations to debug a bottleneck. Some logs track the use of individual resources, producing a time series. Resource-usage statistics often come in the form of cumulative use per time period (for example, *b* bits transmitted in the last minute). One might use bandwidth data to characterize network or disk performance, page swaps to characterize memory effectiveness, or CPU utilization to characterize load-balancing quality.

Like the debugging case, performance logs must be interpreted in context. Two types of contexts are especially useful in performance analysis: the environment in which the performance number occurs and the workload of the system.

Performance problems are often caused by interactions between components, and to reveal these interactions you may have to synthesize information from heterogeneous logs generated by multiple sources. Synthesis can be challenging. In addition to heterogeneous log formats, components in distributed systems may disagree on the exact time, making the precise ordering of events across multiple components impossible to reconstruct. Also, an event that is benign to one component (for example, a log flushing to disk) might cause serious problems for another (for example, because of the I/O resource contention). As the component causing the problem is unlikely to log the event, it may be hard to capture this root cause. These are just a few of the difficulties that emerge.

One approach to solving this problem is to compute *influence*, which infers relationships between components or groups of components by looking for surprising behavior that is correlated in time.[10] For example, bursty disk writes might correlate in time with client communication errors; a sufficiently strong correlation suggests some shared influence between these two parts of the system. Influence can quantify the interaction between components that produce heterogeneous logs, even when those logs are sparse, incomplete, and without known semantics and even when the mechanism of the interaction is unknown. Influence has been applied to production systems ranging from autonomous vehicles such as Stanley[13] (where it helped diagnose a dangerous swerving bug[10]) to supercomputers such as BlueGene/L[1] (where it was able to analyze logs from more than 100,000 components in real time[9]).

Methods that trace a message or request as it is processed by the system are able to account for the order of events and the impact of workload. For example, requests of one type might be easily serviceable by cached data, while requests of another type might not be. Such tracing methods often require supporting instrumentation but can be useful for correctness debugging in addition to understanding performance.

A salient challenge in this area is the risk of influencing the measurements by the act of measuring. Extensive logging that consumes resources can complicate the task of accounting for how those resources are used in the first place. The more we measure, the less accurately we will understand the performance characteristics of the system. Even conservative tracing mechanisms typically introduce unacceptable overhead in practice.

One approach to reduce the performance impact of logging is to sample. The danger is that sampling may miss rare events. If you have millions or even billions of sampled instances of the same program running, however, you may be able to maintain a low sampling rate while still capturing rare events.

An efficient implementation of sampling techniques requires the ability to turn individual log sites on and off without restarting execution. Older systems such as DTrace require statically instrumented log sites.[2] Recent advances in program rewriting can be used to instrument arbitrary sites in program binaries at runtime. One recent effort in this direction is Fay, a platform for the collection, processing, and analysis of software execution traces[3] that allows users to specify the events they want to measure, formulated as queries in a declarative language; Fay then inserts dynamic instrumentation into the running system, aggregates the measurements, and provides analysis mechanisms, all specific to those queries. When applied to benchmark codes in a distributed system, Fay showed single-digit percentage overheads. Dynamic program rewriting combined with sampling-based logging will likely be a key solution to problems requiring detailed logs at scale.

Security

Logs are also used for security applications, such as detecting breaches or misbehavior, and for performing postmortem inspection of security incidents. Depending on the system and the threat model, logs of nearly any kind might be amenable to security analysis: logs related to firewalls, login sessions, resource utilization, system calls, network flows, and so on.

Intrusion detection often requires reconstructing sessions from logs. Consider an example related to intrusion detection—that is, detecting unauthorized access to a system (see the figure here). When a user logs into a machine remotely via SSH, that machine generates log entries corresponding to the login event. On Mac OS X, these look like the messages as depicted in the accompanying figure (timestamp and hostname omitted) that show a user named user47 accessing the machine interactively from a specific IP address and port number.

Common sense says these logout messages match the previous login messages because the hexadecimal session numbers match (0x3551e2); we know the second of these lines, which does not include the session number, is part of the logout event only because it is sandwiched between the other two. There is nothing syntactic about these lines that would reveal, a priori, that they are somehow associated with the lines generated at login, let alone each other.

In other words, each message is evidence of multiple semantic events, including the following: the execution of a particular line of code, the creation or destruction of an SSH session, and the SSH session as a whole.

A log analyst interested in security may then ask the deceptively simple question: Does this SSH session constitute a security breach?

The answer may depend on a number of factors, among them: Have there been an abnormally large number of failed login attempts recently? Is the IP address associated with user47 familiar? Did user47 perform any suspicious actions while the session was active? Is the person with username user47 on vacation and thus should not be logging in?

Note that only some of these questions can be answered using data in the logs. You can look for a large number of failed login attempts that precede this session, for example, but you cannot infer user47's real identify, let alone his or her vacation schedule. Thus, a particular analysis works on logs that are commensurate with the type of attack they wish to detect; more generally, the power of an analysis is limited by the information in the logs.

Log analysis for security may be signature based, in which the user tries to detect specific behaviors that are known to be malicious; or anomaly based, in which the user looks for deviation from typical or good behavior and flags this as suspicious. Signature methods can reliably detect attacks that match known signatures, but are insensitive to attacks that do not. Anomaly methods, on the other hand, face the difficulty of setting a threshold for calling an anomaly suspicious: too low, and false alarms make the tool useless; too high, and attacks might go undetected.

Security applications face the distinguishing challenge of an adversary. To avoid the notice of a log-analysis tool, an adversary will try to behave in such a way that the logs generated during the attack look—exactly or approximately—the same as the logs generated during correct operation. An analysis cannot do much about incomplete logs. Developers can try to improve logging coverage,[16] making it more difficult for adversaries to avoid leaving evidence of their activities, but this does not necessarily make it easier to distinguish a "healthy" log from a "suspicious" one.

The first three messages.

```
sshd[12109]: Accepted keyboard-interactive/pam for user47 from
171.64.78.25 port 49153 ssh2
com.apple.SecurityServer[22]: Session 0x3551e2 created
com.apple.SecurityServer[22]: Session 0x3551e2 attributes 0x20
…
com.apple.SecurityServer[22]: Session 0x3551e2 dead
com.apple.SecurityServer[22]: Killing auth hosts
com.apple.SecurityServer[22]: Session 0x3551e2 destroyed
```

Prediction

Log data can be used to predict and provision for the future. Predictive models help automate or provide insights for resource provisioning, capacity planning, workload management, scheduling, and configuration optimization. From a business viewpoint, predictive models can guide marketing strategy, ad placement, or inventory management.

Some analytical models are built and honed for a specific system. Experts manually identify dependencies and relevant metrics, quantify the relationships between components, and devise a prediction strategy. Such models are often used to build simulators that replay logs with anticipated workload perturbations or load volumes in order to ask what-if questions. Examples of using analytical models for performance prediction exist on I/O subsystems, disk arrays, databases, and static Web servers. This approach has a major practical drawback, however, in that real systems change frequently and analysis techniques must keep up with these changes.

Although the modeling techniques may be common across various systems, the log data mined to build the model, as well as the metrics predicted, may differ. For example, I/O subsystem and operating-system instrumentation containing a timestamp, event type, CPU profile, and other per-event metrics can be used to drive a simulator to predict I/O subsystem performance. Traces that capture I/O request rate, request size, run count, queue length, and other attributes can be leveraged to build analytical models to predict disk-array throughput.

Many analytical models are single tier: one model per predicted metric. In other scenarios a hierarchy of models is required to predict a single performance metric, based on predictions of other performance metrics. For example, Web server traces—containing timestamps, request type (GET vs. POST), bytes requested, URI, and other fields—can be leveraged to predict storage response time, storage I/O, and server memory. A model to predict server response time under various load conditions can be composed of models of the storage metrics and server memory. As another example, logs tracking record accesses, block accesses, physical disk transfers, throughputs, and mean response times can be used to build multiple levels of queuing network models to predict the effect of physical and logical design decisions on database performance.

One drawback of analytical models is the need for system-specific domain knowledge. Such models cannot be seamlessly ported to new versions of the system, let alone to other systems. As systems become more complex, there is a shift toward using statistical models of historical data to anticipate future workloads and performance.

Regression is the simplest statistical modeling technique used in prediction. It has been applied to performance counters, which measure execution time and memory subsystem impact. For example, linear regression applied to these logs was used to predict execution time of data-partitioning layouts for libraries on parallel processors, while logistic regression was used to predict a good set of compiler flags. CART (classification and regression trees) used traces of disk requests specifying arrival time, logical block number, blocks requested, and read/write type to predict the response times of requests and workloads in a storage system.

Both simple regression and CART models can predict a single metric per model. Performance metrics, however, often have interdependencies that must each be predicted to make an informed scheduling or provisioning decision. Various techniques have been explored to predict multiple metrics simultaneously. One method adapts canonical correlation analysis to build a model that captures interdependencies between a system's input and performance characteristics, and leverages the model to predict the system's performance under arbitrary input. Recent work used KCCA (kernel canonical correlation analysis) to model a parallel database system and predict execution time, records used, disk I/Os, and other such metrics, given query characteristics such as operators used and estimated data cardinality.[6] The same technique was adapted to model and predict performance of map-reduce jobs.[5]

Predictive models often provide a range of values rather than a single number; this range sometimes represents a confidence interval, meaning the true value is likely to lie within that interval.

Although these techniques show the power of statistical learning techniques for performance prediction, their use poses some challenges.

Extracting feature vectors from events logs is a nontrivial, yet critical, step that affects the effectiveness of a predictive model. Event logs often contain non-numeric data (for example, categorical data), but statistical techniques expect numeric input with some notion of distributions defined on the data. Converting non-numeric information in events into meaningful numeric data can be tedious and requires domain knowledge about what the events represent. Thus, even given a prediction, it can be difficult to identify the correct course of action.

Predictive models often provide a range of values rather than a single number; this range sometimes represents a confidence interval, meaning the true value is likely to lie within that interval. Whether or not to act on a prediction is a decision that must weigh the confidence against the costs (that is, whether acting on a low-confidence prediction is better than doing nothing). Acting on a prediction may depend on whether the log granularity matches the decision-making granularity. For example, per-query resource-utilization logs do not help with task-level scheduling decisions, as there is insufficient insight into the parallelism and lower-level resource-utilization metrics.

Reporting and Profiling

Another use for log analysis is to profile resource utilization, workload, or user behavior. Logs that record characteristics of tasks from a cluster's workload can be used to profile resource utilization at a large data center. The same data might be leveraged to understand inter-arrival times between jobs in a workload, as well as diurnal patterns.

In addition to system management, profiling is used for business analytics. For example, Web-server logs characterize visitors to a Web site, which can yield customer demographics or conversion and drop-off statistics. Web-log analysis techniques range from simple statistics that capture page popularity trends to sophisticated time-series methods that describe access patterns across multiple user sessions. These insights inform marketing initiatives, content hosting, and resource provisioning.

A variety of statistical techniques have been used for profiling and reporting on log data. Clustering algorithms such as *k*-means and hierarchical clustering group similar events. Markov chains have been used for pattern mining where temporal ordering is essential.

Many profiling and alerting techniques require hints in the form of expert knowledge. For example, the *k*-means clustering algorithm requires the user either to specify the number of clusters (*k*) or to provide example events that serve as seed cluster centers. Other techniques require heuristics for merging or partitioning clusters. Most techniques rely on mathematical representations of events, and the results of the analysis are presented in similar terms. It may then be necessary to map these mathematical representations back into the original domain, though this can be difficult without understanding the log semantics.

Classifying log events is often challenging. To categorize system performance, for example, you may profile CPU utilization and memory consumption. Suppose you have a performance profile for high CPU utilization and low memory consumption, and a separate profile of events with low CPU utilization and high memory consumption; when an event arrives containing low CPU utilization and low memory consumption, it is unclear to which of the two profiles (or both) it should belong. If there are enough such events, the best choice might be to include a third profile. There is no universally applicable rule for how to handle events that straddle multiple profiles or how to create such profiles in the first place.

Although effective for grouping similar events and providing high-level views of system behavior, profiles do not translate directly to operationally actionable insights. The task of interpreting a profile and using it to make business decisions, to modify the system, or even to modify the analysis, usually falls to a human.

Profiles do not translate directly to operationally actionable insights. The task of interpreting a profile and using it to make business decisions, to modify the system, or even to modify the analysis, usually falls to a human.

Logging Infrastructures

A logging infrastructure is essential for supporting the variety of applications described here. It requires at least two features: log generation and log storage.

Most general-purpose logs are unstructured text. Developers use `printf` and string concatenations to generate messages because these primitives are well understood and ubiquitous. This kind of logging has drawbacks, however. First, serializing variables into text is expensive (almost 80% of the total cost of printing a message). Second, the analysis needs to parse the text message, which may be complicated and expensive.

On the storage side, infrastructures such as syslog aggregate messages from network sources. Splunk indexes unstructured text logs from syslog and other sources, and it performs both real time and historical analytics on the data. Chukwa archives data using Hadoop to take advantage of distributed computing infrastructure.[11]

Choosing the right log-storage solution involves the following trade-offs:

- Cost per terabyte (upfront and maintenance)
- Total capacity
- Persistence guarantees
- Write access characteristics (for example, bandwidth and latency)
- Read access characteristics (random access vs. sequential scan)
- Security considerations (access control and regulation compliance)
- Integration with existing infrastructure

There is no one-size-fits-all policy for log retention. This makes choosing and configuring log solutions a challenge. Logs that are useful for business intelligence are typically considered more important than debugging logs and thus are kept for a longer time. In contrast, most debug logs are stored for as long as possible but without any retention guarantee, meaning they may be deleted under resource pressure.

Log-storage solutions are more useful when coupled with alerting and reporting capabilities. Such infrastructures can be leveraged for debugging, security, and other system-management tasks. Various log-storage solutions facilitate alerting and reporting, but they leave many open challenges pertaining to alert throttling, report acceleration, and forecasting capabilities.

Conclusion

The applications and examples in this article demonstrate the degree to which system management has become log-centric. Whether used for debugging problems or provisioning resources, logs contain a wealth of information that can pinpoint, or at least implicate, solutions.

Although log-analysis techniques have made much progress recently, several challenges remain. First, as systems become increasingly composed of many, often distributed, components, using a single log file to monitor events from different parts of the system is difficult. In some scenarios logs from entirely different systems must be cross-correlated for analysis. For example, a support organization may correlate phone-call logs with Web-access logs to track how well the online documentation for a product addresses frequently asked questions and how many customers concurrently search the online documentation during a support call. Interleaving heterogeneous logs is seldom straightforward, especially when timestamps are not synchronized or present across all logs and when semantics are inconsistent across components.

Second, the logging process itself requires additional management. Controlling the verbosity of logging is important, especially in the event of spikes or potential adversarial behavior, to manage overhead and facilitate analysis. The logging mechanism should also not be a channel to propagate malicious activity. It remains a challenge to minimize instrumentation overhead while maximizing information content.

A third challenge is that although various analytical and statistical modeling techniques can mine large quantities of log data, they do not always provide actionable insights. For example, statistical techniques could reveal an anomaly in the workload or that the system's CPU utilization is high but not explain what to do about it. The interpretation of the information is subjective, and whether the information is actionable or not depends on many factors. It is important to investigate techniques that trade off efficiency, accuracy, and actionability.

There are several promising research directions. Since humans will likely remain a part of the process of interpreting and acting on logs for the foreseeable future, advances in visualization techniques should prove worthwhile.

Program analysis methods, both static and dynamic, promise to increase our ability to automatically characterize the interactions and circumstances that caused a particular sequence of log messages. Recent work aims to modify existing instrumentation so that logs are either more amenable to various kinds of analysis or provide more comprehensive information. Although such modifications are not always possible, insights into how to generate more useful logs are often accompanied by insights in how to analyze existing logs. Mechanisms to validate the usefulness of log messages would improve log quality, making log analysis more efficient.

As many businesses become increasingly dependent on their computing infrastructure—not to mention businesses where the computing infrastructure or the services they provide are the business itself—so does the importance of this relationship. We have seen a rise in tools that try to infer how the system influences users: how latency affects purchasing decisions; how well click patterns describe user satisfaction; and how resource-scheduling decisions change the demand for such resources. Conversely, recent work suggests that user activity can be useful for system debugging. Further exploration of the relationships between user behavior (workload) and system behavior may prove useful for understanding what logs to use, when, and for what purpose.

These research directions, as well as better logging standards and best practices, will be instrumental in improving the state of the art in log analysis. C

Related articles on queue.acm.org

Modern Performance Monitoring
Mark Purdy
http://queue.acm.org/detail.cfm?id=1117404

Network Forensics
Ben Laurie
http://queue.acm.org/detail.cfm?id=1016982

The Pathologies of Big Data
Adam Jacobs
http://queue.acm.org/detail.cfm?id=1563874

References

1. BlueGene/L Team. An overview of the BlueGene/L Supercomputer. *IEEE Supercomputing and IBM Research Report* (Nov. 2002).
2. Cantrill, B.M., Shapiro, M.W. and Leventhal, A.H. Dynamic instrumentation of production systems. Usenix 2004 Annual Technical Conference (Boston, MA, June 2004); http://www.usenix.org/event/usenix04/tech/general/full_papers/cantrill/cantrill.pdf.
3. Erlingsson, Ú., Peinado, M., Peter, S., Budiu and M. Fay: Extensible distributed tracing from kernels to clusters. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, Cascais, Portugal (Oct. 2011); http://research.google.com/pubs/archive/37199.pdf.
4. Fonseca, R., Porter, G., Katz R., Shenker, S. and Stoica, I. X-Trace: A pervasive network-tracing framework. *Usenix Symposium on Networked Systems Design and Implementation* (Cambridge, MA , Apr. 2007).
5. Ganapathi, A., Chen, Y., Fox, A., Katz, R. H. and Patterson, D. A. Statistics-driven workload modeling for the cloud. Workshop on Self-Managing Database Systems at ICDE (2010), 87–92.
6. Ganapathi, A., Kuno, H. A., Dayal, U., Wiener, J. L., Fox, A., Jordan, M. I. and Patterson, D. A. Predicting multiple metrics for queries: Better decisions enabled by machine learning. *International Conference on Data Engineering* (2009) 592–603.
7. Gautam, A. and Stoica, I. ODR: output-deterministic replay for multicore debugging. *ACM Symposium on Operating System Principles* (2009), 193–206.
8. Nguyen, X., Huang, L. and Joseph, A. Support vector machines, data reduction, and approximate kernel matrices. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases* (2008), 137–153.
9. Oliner, A.J. and Aiken, A. Online detection of multi-component interactions in production systems. In *Proceedings of the International Conference on Dependable Systems and Networks* (Hong Kong, 2011); http://adam.oliner.net/files/oliner_dsn_2011.pdf.
10. Oliner, A.J., Kulkarni, A.V. and Aiken, A. Using correlated surprise to infer shared influence. In *Proceedings of the International Conference on Dependable Systems and Networks* (Chicago, IL, 2010), 191–200; http://adam.oliner.net/files/oliner_dsn_2010.pdf.
11. Rabkin, A. and Randy, K. Chukwa: A system for reliable large-scale log collection. *USENIX Conference on Large Installation System Administration* (2010), 1–15.
12. Sigelman, B., Barroso, L., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S. and Shanbhag, C. Dapper, a large-scale distributed systems tracing infrastructure. Google Technical Report; http://research.google.com/archive/papers/dapper-2010-1.pdf.
13. Thrun, S. et al. Stanley: The robot that won the DARPA Grand Challenge. *Journal of Field Robotics 23*, 9 (2006), 661–692.
14. Xu, M. et al. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual International Symposium on Computer Architecture* (San Diego, CA, June 2003).
15. Xu, W., Huang, L., Fox, A., Patterson, D. and Jordan, M. Detecting large-scale system problems by mining console logs. In *Proceeding of the 22nd ACM Symposium on Operating Systems Principles* (Big Sky, MT, Oct. 2009).
16. Yuan, D., Zheng, J., Park, S., Zhou, Y. and Savage, S. Improving software diagnosability via log enhancement. In *Proceedings of Architectural Support for Programming Languages and Operating Systems* (Newport Beach, CA, Mar. 2011); http://opera.ucsd.edu/paper/asplos11-logenhancer.pdf.

Adam Oliner is a postdoctoral scholar in electrical engineering and computer sciences at UC Berkeley, working with Ion Stoica and the AMP (Algorithms, Machine and People) Lab.

Archana Ganapathi is a research engineer at Splunk, where she focuses on large-scale data analytics. She has spent much of her research career analyzing production datasets to model system behavior.

Wei Xu is a software engineer at Google, where he works on Google's debug logging and monitoring infrastructure. His research interest is in cluster management and debugging.