Online Detection of Multi-Component Interactions in Production Systems

Adam J. Oliner and Alex Aiken
Department of Computer Science*
Stanford University
{oliner, aiken}@cs.stanford.edu

***Abstract*—We present an online, scalable method for inferring the interactions among the components of large production systems. We validate our approach on more than 1.3 billion lines of log files from eight unmodified production systems, showing that our approach efficiently identifies important relationships among components, handles very large systems with many simultaneous signals in real time, and produces information that is useful to system administrators.**

***Keywords*-System management, statistical correlation, modeling, anomalies, signal compression**

I. Introduction

We are interested in automatic support for understanding large production systems such as supercomputers, data center clusters, and complex control systems. Fundamentally, administrators of such systems need to answer variations of one question: *What parts of the system might affect X?* Most obviously, *X* may be the manifestation of a system bug and the administrator is looking for the cause, but administrators also need to answer questions about what affects resource utilization (e.g., to eliminate performance problems), global or local unexplained behavior (e.g., to decide whether the unexpected behavior is actually a problem or not), and even what aspects of the system should be monitored (with the aim of logging all and only useful data), to give just a few examples. Over a period of several years we have learned, usually the hard way, that there are severe constraints on any solution to this problem:

1) *Lack of specification.* In practice there is no description of the correct behavior of the system. In fact, in all the systems we have studied there is not even a list of all the system's components and their interactions—even the administrators are unaware of what is inside some parts of the system (e.g., third-party subsystems may be black boxes). Administrators do have rules of thumb and lists of known bad behaviors that they they watch for; they also realize these lists are incomplete.
2) *Minimally invasive monitoring.* For reasons of cost, performance, and system stability, administrators are almost without exception unwilling to disturb the inner workings of system components for the purposes of better monitoring. It is often possible to add new logging of inputs and outputs to components, but even that normally must be justified as cost-effective for answering some important question that cannot be answered using existing logs.
3) *Rapid turnaround.* Answers to some of the most important questions are only useful if they can be computed in real-time. For example, administrators would like to set standing queries that trigger an alarm when the system first strays into a pattern of behavior that is known to likely lead to severe problems or a crash.

In previous work we have addressed problems (1) and (2). We assume only that some subset of the components have logs with time-stamped entries (every system we have seen satisfies this requirement). These logs are converted into time-varying signals that are correlated, possibly with a time delay. The strength of the correlation and direction of any delays allow administrators to answer many useful queries about how and when various parts of the system influence each other. All of this computation, however, is offline [25], [27].

The primary contribution of this paper is an online method for analyzing and answering questions about large systems. Our approach retains the idea of computing correlations and delays between component signals, but the semantic and performance requirements of an online solution result in a different design. In particular, we use a novel combination of online, anytime algorithms that maintain concise models of how components and sets of components are interacting with each other, including the delays or lags associated with those interactions. The types of questions our method is able to answer online without the need for specifications or invasive logging and the scalability of our approach are both new.

A system consists of a set of components, some of which are instrumented to record timestamped log entries. These logs are converted into real-valued functions of time called *anomaly signals*, which encode when measurements differ from typical or expected behavior. The process of converting raw logs into meaningful anomaly signals is how the user encodes what they know about the system as well as what they want to understand. For example, a user might want the anomaly signal to initially highlight an unusual error message and then mute it once the error is understood. System administrators are comfortable with this notion of an exploratory tool, which they can adapt to reflect changes in the system, in their knowledge of it, or in the questions they want to answer. While we discuss

*This work was supported in part by NSF grant CCF-0915766 and the DOE High-Performance Computer Science Fellowship.

this further in Section 3.1, the process of converting raw measurements into anomaly signals is well-studied for a wide variety of data types [11], [13], [15], [20], [25], [27], [39], [40] and this paper focuses instead on how to use anomaly signals to efficiently infer component interactions.

At every time-step or *tick*, we pass the most recent value of every anomaly signal through a two-stage analysis. The first stage compresses the data by finding correlated groups of signals using an online, approximate principal component analysis (PCA) [30]; we call these component groups *subsystems*. This analysis produces a new set of anomaly signals, called *eigensignals*, with one eigensignal corresponding to the behavior of each subsystem; in other words, the behavior of the entire system is summarized using a new, and much smaller, set of signals. The second stage takes these eigensignals, and possibly a small set of additional anomaly signals (see Section III-C1), and looks for lag correlations among them using an online approximation algorithm [34]. Although the eigensignals are mutually uncorrelated by construction, they may be correlated with some lag.

Figure 1 shows an example with three signals taken from a production database (SQL) cluster: `disk` (an aggregated signal corresponding to disk activity), `forks` (corresponding to the average number of forked processes), and `swap` (corresponding to the average number of memory page-ins). The first stage of the analysis, the PCA, automatically finds the correlation between `disk` and `forks` and generates a single eigensignal that summarizes both of the original signals. The second stage of the analysis takes the eigensignal and `swap`'s anomaly signal, plotted in Figure 2, and discovers a correlation: surprising behavior in the subsystem consisting of `disk` and `forks` tends to precede surprising behavior in `swap`. Our analysis, on these and several related signals, helped the system's administrator diagnose a performance bug: a burst of disk swapping coincided with the beginning of a gradual accumulation of slow queries which, over several hours, crossed a threshold and crippled the server. In addition to helping with a diagnosis, our method can give enough warning of the impending collapse for the administrator to take remedial action (see Section V-D2).

We describe our method in Section III and evaluate it using nearly 100,000 signals from eight unmodified production systems (described in Section IV), including four supercomputers, two autonomous vehicles, and two data center clusters. Our results, in Section V, show that we can efficiently and accurately discover correlations and delays in real systems and in real time, and furthermore that this information is operationally valuable.

II. Related Work

There is an extensive body of work on system modeling, especially on inferring the causal or dependency structure of distributed systems. Our method distinguishes itself from most previous work primarily in that we look for correlated behavior called *influence* rather than *dependencies* [2], [10], [35], [41]. Two components share an influence if there is a correlation in

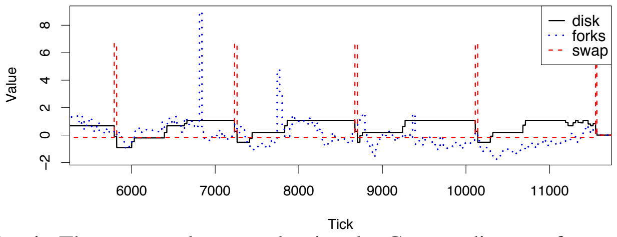

Fig. 1: Three example anomaly signals. Greater distance from zero (average) corresponds to more surprising measurements. The signals `disk` and `forks` are statistically correlated.

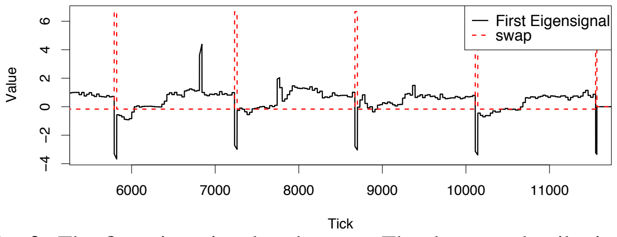

Fig. 2: The first eigensignal and `swap`. The downward spike in the eigensignal consistently happens just before the spike in `swap`.

their deviations from normal behavior; influence is orthogonal to whether or not the components share dependencies. Our previous work has shown that influence is statistically robust to noisy or missing data and captures implicit interactions like resource contention [27]. The main contributions of this paper are in computing both the strength and directionality (time delay) of influence online, scaling to tens of thousands of signals, and applying this tool to several administration tasks.

Previous work on dependency graphs typically assumes that the system can be perturbed (by adding instrumentation or active probing), that the user can specify the desired properties of a healthy system, that the user has access to the source code, or some combination of these (e.g., [22], [36]). We discuss work that falls into each of these classes in greater detail below. In our experience, it is often the case that none of these assumptions hold in practice.

One common thread in dependency modeling work is that the system must be actively perturbed by instrumentation or by probing [4], [5], [8], [9], [33]. Pinpoint [6], [7] and Magpie [3] track communication dependencies with the aim of isolating the root cause of misbehavior; they require instrumentation of the application to tag client requests. In order to determine the causal relationships among messages, Project5 [1] and WAP5 [32] use message traces and compute dependency paths (none of the systems we studied recorded such information). D^{3}S [21] uses binary instrumentation to perform online predicate checks. Other work leverages tight integration of the system with custom instrumentation to improve diagnosability (e.g., the P2 system [36]) or restrict the tool to particular kinds of systems (e.g., MapReduce [29] or wide area networks [12], [18], [19], [42]). Deterministic replay is another common approach [14], [22] but requires supporting instrumentation. For all eight of the production systems we study, we could not apply any of these existing methods, and it was neither possible nor practical for us to add instrumentation.

Some approaches require the user to write predicates indicating what properties should be checked [21], [22], [36]. Pip [31] identifies when communication patterns differ from expectations and requires an explicit specification of those expectations. We have no such correctness predicates, models, or specifications for any of the systems we study. Furthermore, we encountered many instances where it would not have been possible to write a sufficient specification of correct behavior before diagnosing the problem—in other words, knowing what property to check (e.g., creating a model suitable for model checking) was equivalent to understanding the root cause.

Recent work shows how access to source code can facilitate tasks like log analysis [40] and distributed diagnosis [16]. It is worth noting that Xu et al. recently also used principal component analysis in their work [39]; they use it to identify anomalous event patterns rather than finding related groups of real-valued signals. Although our system could be extended to take advantage of access to source code, many systems involve proprietary, third-party, or classified software for which source code is unavailable.

A passive, black-box technique like ours has two potential drawbacks in comparison with using dependencies. First, it can only infer correlation, which does not imply causality. Dependency-based techniques actually track cause and effect and can say with certainty that certain events caused other events to happen. Second, the correlations are only as good as the log data, and less information about fewer components will typically yield fewer and weaker correlations. In practice, however, we have found that administrators already gather enough information for our approach to discover useful correlations and to imply causality using temporal ordering. In Sections V-C and V-D, we demonstrate real situations in which our method provides high-quality, actionable information, despite these potential limitations.

Many interesting problems in systems arise when components are connected or composed in ways not anticipated by their designers [23]. As systems grow in scale, the sparsity of instrumentation and complexity of interactions increases. Our technique infers a broad class of interactions in unmodified production systems, online, using existing instrumentation.

Our method uses an online principal component analysis adapted from SPIRIT [30] and a lag correlation detection algorithm called Enhanced BRAID [34]. We selected these techniques because they make only weak assumptions about the input data and have good performance and scalability characteristics. This paper employs those two methods in several novel ways: using the PCA as a dimensionality reduction to make the lag correlation scalable, analyzing anomaly signals rather than raw data as the input to permit the comparison of heterogeneous components and the encoding of expert knowledge, adding a mechanism for bypassing the PCA stage that we use for standing queries, and, finally, applying these techniques in the context of understanding production systems.

III. Method

Our method takes a difficult problem—understanding the complex relationships among heterogeneous components generating heterogeneous logs—and transforms it into a well-formed and computable problem: understanding the *variance* in a set of signals. The input to our method is a set of signals for which variance corresponds to behavior lacking a satisfactory explanation. The first stage of our method tries to explain the variance of one signal using the variance of other signals; the standard technique for doing this is called principal component analysis (PCA). However, PCA will miss signals that co-vary with some delay or lag. The second stage of our method identifies such lagged correlations. Furthermore, we show how to encode and answer many natural questions about a system in terms of time varying signals.

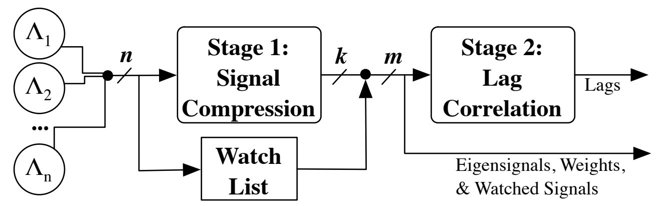

Fig. 3: Our method takes n anomaly signals and passes them to a signal compression stage (performed using principal component analysis). It then takes the resulting k eigensignals and those on the watch list and passes these m signals to a lag correlation detector. The listed output values are all available at any time.

Consider a system of components in which a subset of these components are generating timestamped measurements that describe their behavior. These measurements are represented as real-valued functions of time called *anomaly signals* (see Section III-A). Our method consists of two stages that are pipelined together: (i) an online PCA that identifies the contributions of each signal to the behavior of the system and identifies groups of components with mutually correlated behavior called *subsystems* (see Section III-B) and (ii) an online lag correlation detector, which determines whether any of these subsystems are, in turn, correlated with each other when shifted in time (see Section III-C). Figure 3 provides an overview of our approach.

A. Anomaly Signals

The input to our method is timestamped measurements from components. The measurements from a particular component are used to construct an *anomaly signal*. The value of an anomaly signal at a given time represents how unusual or surprising the corresponding measurements were: the further from the signal's average value, the more surprising.

As with any abstraction, anomaly signals are used to hide details of the underlying data that are irrelevant for answering a particular question. Thus, there is no single "correct" anomaly signal, as any feature of the log may be useful for answering some question. The abstraction may only lessen, rather than remove, unwanted characteristics and may unintentionally mute important signals, but the purpose of the anomaly signal abstraction is to highlight the behaviors we wish to understand, especially when and where they are occurring in the system.

Some measurements require a processing step to make them numerical (see Section III-A1). In the absence of any special knowledge about the system or the mechanisms that generated the data, we have found that anomaly signals based on statistical properties (e.g., the frequency of particular words in a textual log) work quite well.

Administrators do not typically have a complete specification of expected behavior: systems are too complicated and change too frequently for such a specification to be constructed or maintained. Instead, they often have short lists of rules about what kinds of events in the logs are important. Anomaly signals allow them to encode this information (see Section III-A1a).

A single physical or logical component may produce multiple signals, each of which has an associated name. For example, a server named `host1` may be recording bandwidth measurements as well as syslog messages, so the corresponding signals might be named `host1-bw` and `host1-syslog`, respectively. A single measurement stream may be used to construct multiple anomaly signals: a text log might have one signal for how unusual the messages are, overall, and another signal for the presence or absence of a particular message.

Conversely, we don't assume that all components have at least one signal, and every real system we have examined has multiple components that are uninstrumented. In fact, some components were even unknown to the administrators. For this reason, we can't use techniques that assume instrumentation for and knowledge of all components in the system.

1) Derived Signals: Non-numerical data like log messages or categorical states must be converted into anomaly signals. This process is well-studied for certain types of data, e.g., unstructured or semi-structured text logs [39], [40]. We use the Nodeinfo algorithm [26] for textual logs and an information-theoretic timing-based model [27] for the embedded systems (autonomous vehicles), as both algorithms highlight irregularities in the data without requiring a deep understanding of it.

Users may optionally preprocess numerical signals to encode what aspects of the measurements are interesting and which are not. For example, daily traffic fluctuations may increase variance, but this is not surprising and may be filtered out of the anomaly signal. We apply no such filtering.

Although numerical signals can be used directly and there are existing tools for getting anomaly signals out of common data types, the more expert knowledge the user applies to generate anomaly signals from the data, the more relevant our results. In particular, the administrators of our systems maintained lists of log message patterns that they believe correspond to important events and they had a general understanding of system topology and functionality; we now discuss how that knowledge can be used to generate additional anomaly signals from the existing log data.

a) Indicator Signals: A user can encode knowledge of interesting log messages using a signal that indicates whether a predicate (e.g., a specific component generated a message containing the string `ERR` in the last five minutes) is true or false [25]. Although this is the simplest way to encode expert knowledge about a log, indicator signals have proven to be both flexible and powerful. Section V-C4 gives an example of how indicator signals can elucidate system-wide patterns.

b) Aggregate Signals: A user can encode knowledge of system topology (e.g., a set of signals are all generated by components in a single machine rack) by computing the time-wise average of those signals. This new signal represents the aggregate behavior of the original signals; the time-average of correlated signals will tend to look like the constituent signals while the average of uncorrelated or anti-correlated signals will tend toward a flat line. This has been shown to be a useful way for the user to describe functionally- or topologically-related sets of signals [25], and we see in Section V-C that these aggregate signals often summarize important behaviors.

B. Stage 1: Signal Compression

A system may have thousands of anomaly signals, so being able to efficiently summarize them using only a small number of signals, with minimal loss of information, is valuable to users of our approach and sometimes necessary to achieve adequate online performance.

To compress the anomaly signals with minimal loss of information, the first stage of our analysis performs an approximate, online principal component analysis (PCA) [30]. This stage takes the n anomaly signals, where n may be large, and represents them as a small number k of new signals that are linear combinations of the original signals. These new signals, which we call *eigensignals*, are computed so that they capture or describe as much of the variance in the original data as possible; the parameter k is set to be as large as computing resources allow to minimize information loss. This stage is online, any-time, single-pass, and does not require any sliding windows or buffering.

Although we refer the reader to the original paper for details [30], we include a brief summary here for completeness. The PCA maintains, for each eigensignal, a vector of *weights* of length n, where n is the number of anomaly signals. At each *tick* (time step), for each eigensignal, a vector containing the most recent value of each anomaly signal is projected onto the weight vector to produce a value for the eigensignal. The eigensignals and weights are then used to reconstruct an approximation of the original n signals. A check ensures the resulting reconstruction has an *energy* that is sufficiently close to that of the original signals; if not, the weights are adjusted so that they "track" the anomaly signals. The time and space complexity of this method on n signals and k eigensignals is $O(nk)$. An eigensignal and its weights define a *behavioral subsystem*: a linear combination of related signals.

Recall the example from Section I. The first stage groups `disk` and `forks` in the same subsystem, and in fact these two signals are highly correlated. At this point, however, there is no apparent relationship with the `swap` component. Note that although PCA will tend to group correlated signals because this efficiently explains variance, two signals being in the same subsystem does not imply that they are highly correlated. This is easily checked, though we omit the details. It has been our experience that the signals with significant weight in a subsystem are all well-correlated, which is also the justification for picking the most heavily weighted signal in a subsystem as the *representative* of that subsystem (see Section V-C3).

1) Decay: The PCA stage takes an optional parameter that causes old measurements to be gradually forgotten, so the subsystems will weight recent data more than older data. This *decay* parameter is set to 1.0 by default, which means all historical data is considered equally in the analysis. Previous work used a decay parameter of 0.96 [30]. In our experiments, we say 'no decay' to indicate a decay value of 1.0 and 'decay' to indicate 0.96. Note, however, that we do not explicitly retain historical data, in either case.

Decay is useful for more closely tracking recent changes (see Section V-C5) and for studying those changes over time (see Section V-C1); if needed, an instance of the compression stage with decay can be run in parallel to one without. We use no decay except where otherwise indicated.

C. Stage 2: Lag Correlation

The first stage of our method extracts correlations among signals that are temporally aligned, but delayed effects or clock skews may cause correlations to be missed. The second stage performs an approximate, online search for signals correlated with a *lag* [34]; that is, signals that are correlated when one is shifted in time relative to the other.

Again, we describe the lag correlation stage here for completeness, but refer the reader to the original paper for details [34]. The cross-correlation between two signals gives the correlation coefficients for different lags; the cross-correlation can be updated incrementally, while retaining only a set of sufficient statistics about the two input signals. To reduce the running time, lag is computed only at a subset of lag values, chosen so that smaller lags are computed more densely than larger lags. To reduce space consumption, lags are computed on smoothed approximations of the original signals. These optimizations yield asymptotic speedups and typically introduce little to no error (see Section V-D3). The running time, per tick, is $O(m^2)$, where m is the number of signals. The space complexity is $O(m^2 \log t)$, where t is the number of ticks.

One of the insights of our approach is that, without first reducing the dimensionality of the problem, large systems would generate too many signals for lag correlation to be practical; one of the primary purposes of the PCA computation is to perform this dimensionality reduction. Once the problem is reduced to eigensignals and perhaps a small set of other signals (see Section III-C1), lag correlation can often be computed more quickly than the PCA (see Section V-A). In other words, the first stage of our method ensures $m << n$ and makes lag correlation practical for large systems.

Recall the example from Section I. The lag correlation stage finds a temporal relationship between the subsystem consisting of `disk` and `forks` and the component `swap`, specifically that anomalies in the former tend to precede those in the latter.

1) Watch List: The *watch list* is a small set of signals that, in addition to the eigensignals, will be checked for lag correlations. These signals bypass the compression stage, which enables us to ask questions (standing queries) about

System	Comps	Log Lines	Time Span
Blue Gene/L	131,072	4,747,963	215:00:00:00
Thunderbird	9024	211,212,192	244:00:00:00
Spirit	1028	272,298,969	558:00:00:00
Liberty	445	265,569,231	315:00:00:00
Mail Cluster	33	423,895,499	10:00:05:00
Junior	25	14,892,275	05:37:26
Stanley	16	23,465,677	09:06:11
SQL Cluster	9	116,785,525	09:00:47:00

TABLE I: The seven unmodified production system logs used in our case studies. The 'Comps' column indicates the number of logical components with instrumentation; some did not produce logs. Real time is given in days:hours:minutes:seconds.

specific signals and to associate results with specific components. There are several ways for a signal to end up on the watch list: manual addition (e.g., a user complains that a certain machine has been misbehaving), automatic addition by rule (e.g., if the temperature of some component exceeds a threshold), or automatic by selecting *representatives* for the subsystems (see Section V-C3). A subsystem's representative signal is the anomaly signal with the largest absolute weight in the subsystem that is not the representative of an earlier (stronger) subsystem. In our experiments, we automatically seed the watch list with the representative of each subsystem.

D. Output

The output of our method is the behavioral subsystems, their behavior over time as eigensignals, and lag correlations between those eigensignals and signals on the watch list. The first stage produces k eigensignals and their weights. The second stage produces a list of pairs of signals from among the eigensignals and those on the watch list that have a lag correlation, as well as the values of those lags and correlations. This output is available at any time during execution.

IV. Systems

We evaluate our method on data from eight production systems: four supercomputers, two data center clusters, and two autonomous vehicles. Table I summarizes these systems and logs, described in Sections IV-A–IV-C and elsewhere [24], [25], [26], [28], [37], [38]. For this wide variety of systems—without modifying, instrumenting, or perturbing them in any way—our method builds online models of component and subsystem interactions, and we use these results for several system administration tasks.

The focus of this paper is on how to analyze anomaly signals online to identify multi-component interactions, rather than how to generate those anomaly signals in the first place, so—for every system—we use the results of previous work to pick algorithms to convert raw data into anomaly signals and for picking predicates to generate indicator signals [25], [26], [27]. These data are summarized in Table II. It has been our experience that the results of our method are not strongly sensitive to choices of these algorithms; for any reasonable choice of anomaly signals, our method tends to group similar components and detect similar lags.

System	Ticks	Tick=	Signals	Agg.	Ind.
Blue Gene/L	2985	1 hr	69,087	67	245
Thunderbird	3639	1 hr	18,395	7	13,573
Spirit	11,193	1 hr	4094	7	3569
Liberty	5362	1 hr	372	4	124
Mail Cluster	14,405	1 min	139	4	102
Junior	488,249	0.04 s	25	0	0
Stanley	821,897	0.04 s	16	0	0
SQL Cluster	13,007	1 min	368	26	34

TABLE II: Summary of the anomaly signals for this study. We omit ticks in which no logs were generated. The 'Signals' column indicates the total number of anomaly signals, which includes the aggregate ('Agg.') and indicator ('Ind.') signals.

A. *Supercomputers*

We use publicly-available logs from supercomputers that were in production use at national laboratories [37]. These four systems, named Liberty, Spirit, Thunderbird, and Blue Gene/L (BG/L), vary in size by several orders of magnitude, ranging from 512 processors in Liberty to 131,072 processors in BG/L. The logs were recorded during production use of these systems and we make no modifications to them, whatsoever. An extensive study of these logs can be found elsewhere [28]. The log messages below were generated consecutively by node `sn313` of the Spirit supercomputer:

```
Jan 1 01:18:56 sn313/sn313 kernel: GM: There are 1
    active subports for port 4 at close.
Jan 1 01:19:00 sn313/sn313 pbs_mom: task_check,
    cannot tm_reply to 7169.sadmin2 task 1
```

We use an algorithm based on the frequency of terms in log messages [26] to generate anomaly signals from the raw data. This is a reasonable algorithm to use if nothing is known of the semantics of the log messages; less frequent symbols carry more information than frequent ones.

We generate indicator signals corresponding to known alerts in the logs [26] using a process described in detail elsewhere [25]. These signals indicate when the system or specific components generate a message matching a regular expression that is known to correspond to interesting behavior. For example, one message generated by Blue Gene/L reads, in part:

```
excessive soft failures, consider replacing the card
```

The administrators are aware that this so-called `DDR_EXC` alert indicates a problem. We generate one anomaly signal, called `DDR_EXC`, that is high whenever any component of BG/L generates this alert; for each such component (e.g., `node1`), there are also corresponding anomaly signals that are high whenever that component generates the alert (called `node1/DDR_EXC`) and whenever that component generates any alert (called `node1/*`).

We also generate aggregate signals for the supercomputers based on functional or topological groupings provided by the administrators. For example, Spirit has aggregate signals for the administrative nodes (`admin`), the compute nodes (`compute`), and the login nodes (`login`). For Thunderbird and BG/L, we also generate an aggregate signal for each rack.

B. *Clusters*

We also obtained logs from two clusters at Stanford University: 17 machines of a campus email routing server cluster and 9 machines of a SQL database cluster. Of the 17 mail cluster servers, 16 recorded two types of logs: a sendmail server log and a Pure Message log (a spam and virus filtering application). One system recorded only the mail log. The SQL cluster was unique among the systems we studied in that it recorded (a total of 271) numerical metrics using the Munin resource monitoring tool (e.g., bytes received, threads active, and memory mapped). For example, the following lines are from the memory swap metric:

```
2009-12-05 23:30:00 6.5536000000e+04
2009-12-06 00:00:00 6.3502367774e+04
```

Each such numerical log was used without modification as an anomaly signal. To generate anomaly signals for the non-numeric content of these logs, we use the same term-frequency algorithm as in Section IV-A.

As with the supercomputers, we generate indicator signals for the textual parts of the cluster logs. Unlike the supercomputers, however, there are no known alerts, so we instead look for the strings 'error', 'fail', and 'warn' and name these signals `ERR`, `FAIL`, and `WARN`, respectively. These strings may turn out to be subjectively unimportant, but adding them to our analysis is inexpensive. We also generate aggregate signals based on functional groupings provided by the administrators. For example, the mail cluster has one aggregate signal for the SMTP logs and another for the spam filtering logs; similarly, we aggregate disk-related logs in the SQL cluster into a signal called `disk`, memory-related logs into `memory`, etc.

C. *Autonomous Vehicles*

Stanley is the autonomous diesel-powered Volkswagen Touareg R5 developed at Stanford University that won the DARPA Grand Challenge in 2003 [38]. A modified 2006 Volkswagen Passat wagon named Junior placed second in the subsequent Urban Challenge [24]. These distributed, embedded systems consist of many sensor components (e.g., lasers, radar, and GPS), a series of software components that process and make decisions based on these data, and interfaces with the cars themselves (e.g., steering and braking). In order to permit subsequent replay of driving scenarios, some of the components were instrumented to record inter-process communication. These log messages indicate their source, but not their destination (there are sometimes multiple consumers). We use the raw logs from the Grand Challenge and Urban Challenge, respectively. The following lines are from Stanley's Intertial Measurement Unit (IMU):

```
IMU -0.001320 -0.016830 -0.959640 -0.012786 0.011043
    0.003487 1128775373.612672 rr1 0.046643
IMU -0.002970 -0.015510 -0.958980 -0.016273 0.005812
    0.001744 1128775373.620316 rr1 0.051298
```

In the absence of expert knowledge, we generate anomaly signals based on deviation from what is typical: unusual terms in text-based logs or deviation from the mean for numerical

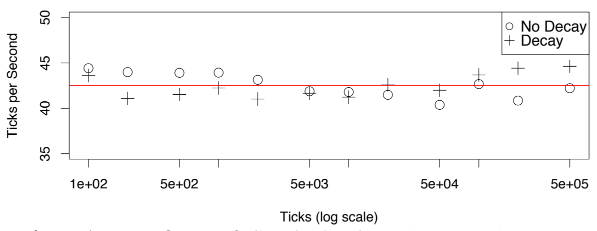

Fig. 4: Using prefixes of Stanley's data ($n = 16$), we see that compression rate is not a function of the number of ticks.

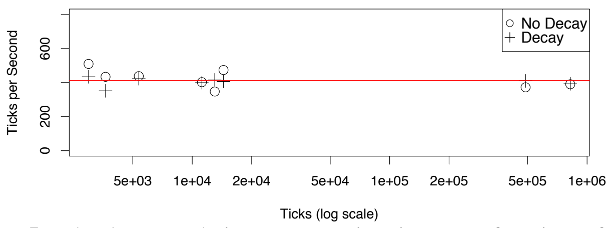

Fig. 5: The lag correlation computation is not a function of the number of ticks ($n = 20$). Each pair of data points corresponds to one of our studied systems.

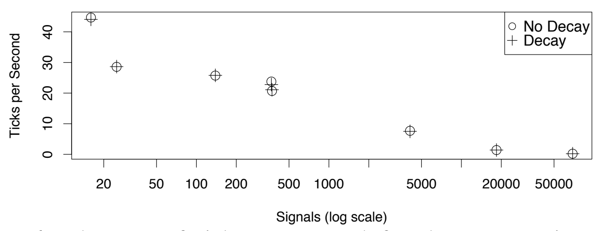

Fig. 6: The rate of ticks per second for the compression stage decreases slowly with the number of signals; autoregressive weighting (decay) has no effect on running time.

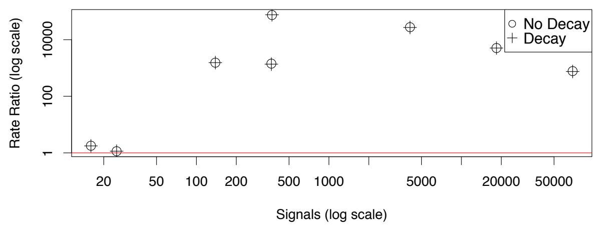

Fig. 7: Although the compression rate decreases with the number of signals, larger systems tend to update measurements less frequently. The ratio between compression rate and measurement generation rate, plotted, shows that the bigger systems are easier to handle than the 25 ticks-per-second data rate of the embedded systems.

logs. Stanley's and Junior's logs contained little text and many numbers, so we instead leverage a different kind of regularity in the logs, namely the interarrival times of the messages. We compute anomaly signals using an existing method based anomalous distributions of message interarrival times [25]. We generate no indicator or aggregate signals for the vehicles.

V. Results

Our results show that we can easily scale to systems with tens of thousands of signals and that we can describe most of a system's behavior with eigensignals that are orders of magnitude smaller than the original data; the behavioral subsystems and lags our method discovers correspond to real system phenomena and have operational value to administrators.

In this paper, we use a static $k = 20$ eigensignals rather than attempt to dynamically adapt this number to match the variance in the data (as suggested elsewhere [30]). It was our experience that such adaptation resulted in overly frequent changes to k. Instead, we set k to the largest value at which the analysis is able to keep up with the rate of incoming data. For the system that generated data at the highest rate (Junior), this number was approximately 20, and we use this value throughout.

As stated in Sections III-B1 and III-C1, we test decay values of 1.0 ('no decay') and 0.96 ('decay') in agreement with previous work, and we automatically seed the watch list with representatives from the subsystems, except where noted.

We performed all experiments on a MacPro with two 2.66 GHz Dual-Core Intel Xeons and 6 GB 667 MHz DDR2 FB-DIMM memory, running Mac OS X version 10.6.4, using a Python implementation of the method.

Section V-A describes the performance of our analysis in terms of time and Section V-B discusses the quality of the results; we focus in these subsections on the mechanisms of the analysis, rather than their applications. Then, in Sections V-C–V-D, we discuss use cases for our method with examples from the data. There are a variety of techniques for visualizing the information produced by our analysis (e.g., graphs [25]); this section focuses instead on the information our method produces and the uses of that information.

A. Performance

Our method is easily able to keep up with the rate of data production for all the systems that we studied.

The performance per tick does not degrade over time. Figures 4 and 5 show processing rate in ticks per second for the signal compression and lag correlation stages, respectively. Across more than three orders of magnitude of ticks, from 100 to around 821,000, there is no change in performance. This is in contrast to the naïve PCA algorithm, whose running time grows linearly with number of ticks.

The compression stage scales well with the number of signals (see Figure 6). For systems with a few dozen components, the entire PCA state can be updated dozens of times per second. Even with 70,000 signals, one tick takes only around 5 seconds. For such larger systems, however, the per-component rate at which instrumentation data is generated tends to be slower, as well. We require the rate of processing to exceed the rate of data generation. As noted above, we chose a number of subsystems that guaranteed this rate ratio was greater than 1 for all the systems we studied. The interesting fact is that for many of the larger systems the ratio was much higher (see Figure 7). In other words, the compression stage is sufficiently fast to handle tens of thousands of signals that update with realistic frequency. In fact, it was Junior, one of the smaller systems, that had the smallest ratio of around 1.14. Junior's 25 anomaly signals were updating 25 times per second.

In the event that a system were to produce data too quickly, either because of the total number of signals or because of the update frequency, we could reduce the number of subsystems

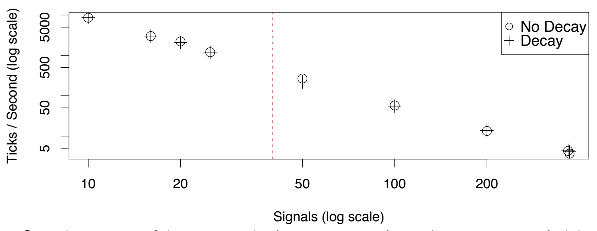

Fig. 8: The rate of lag correlation processing decreases quickly with the number of signals. (Note the log-log scale.) Our method uses eigensignals and a watch list to keep the number of signals small.

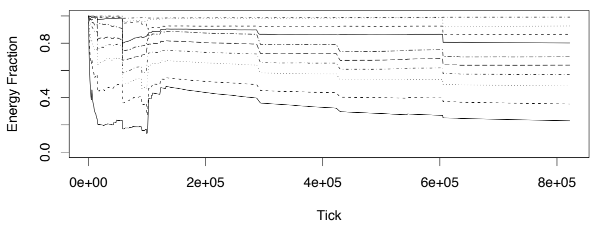

Fig. 9: The cumulative fraction of total energy in Stanley's first k eigensignals. The bottom line shows the energy captured by the first eigensignal; the line above that is for the first two eigensignals, etc.

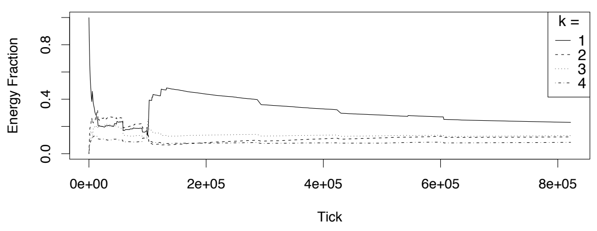

Fig. 10: The incremental additional energy captured by Stanley's k^{th} eigensignal, given the first $k-1$.

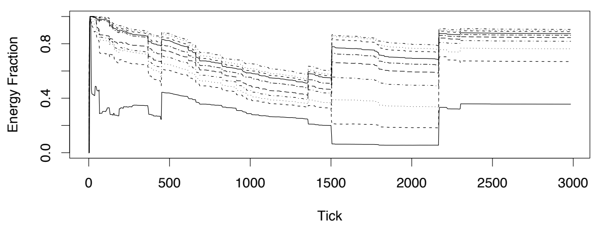

Fig. 11: The cumulative fraction of total energy in BG/L's first k eigensignals. The first ten eigensignals suffice to describe more than 90% of the energy in the system's 69,087 signals.

(k), reduce the size of the watch list, or reduce the anomaly signal sampling rate. This was not necessary for any of our systems. Note that bursts in the raw log data, which can exceed the average message rate by many orders of magnitude, are absorbed by the anomaly signal and do not factor into this discussion of data rate. Furthermore, we believe that future work could parallelize both stages of our analysis, yielding even better performance.

As Figure 8 shows, the lag correlation stage scales poorly with the number of signals. Trying to run it on all 69,087 signals from BG/L, for example, is intractable. Our method skirts this problem by feeding the lag correlation stage only m signals: the eigensignals and signals on the watch list. The vertical line at 40 signals represents the number we use for most of the remaining experiments: 20 eigensignals and 20 representative signals in the watch list. Our method scales to supercomputer-scale systems because $m << n$.

B. Eigensignal Quality

Previous work uses a measure called *energy* to quantify how well the eigensignals describe the original signals [17], [30]. Let $x_{\tau,i}$ be the value of signal i at time τ. The energy E_t at time t is defined as $E_t := \frac{1}{t}\sum_{\tau=1}^{t}\sum_{i=1}^{n} x_{\tau,i}^2$.

By projecting the eigensignals onto the weights, we can reconstruct an approximation of the original n anomaly signals. If the eigensignals are ideal, then the energy of the reconstructed signals will be equal to the energy of the original signals; in practice, using $k << n$ eigensignals and online approximations means that this fraction of reconstruction energy to original energy will be less than one.

Consider the autonomous vehicle, Stanley, which has 16 original signals. Figure 9 shows the energy ratio for the first ten eigensignals; the lowest line is for the first eigensignal only, the line above that represents the first two eigensignals, then the first three, and so on. Figure 10 shows the incremental energy fraction; that is, the line for $k = 3$ shows the amount of increase in the energy fraction over using $k = 2$. Near the beginning of the log, the PCA is still learning about the system's behaviors, so the energy fraction is erratic. Over time, however, the ratio stabilizes. These experiments were without decay, so the energy fractions show how well the compression stage is able to model all the data it has seen so far. The first ten eigensignals are able to model almost 100% of the energy of Stanley's 16 original signals (i.e., almost 38% of the information in the anomaly signals was redundant).

For larger systems, we find more signals tend to be correlated and the number of eigensignals needed per original signal decreases. Consider the cumulative energy fraction plot for BG/L in Figure 11, which shows that the first eigensignal, alone, contains roughly 33% of all of the energy in the system.

Figure 12 shows what fraction of energy is captured by the first k eigensignals as a function of $\frac{k}{n}$. In other words, if we think of the first stage of our method as lossy compression, the figure shows how efficiently we are compressing the data and with what loss of information. For systems like BG/L, with many correlated subsystems, we can describe most of the behavior with a tiny fraction of the original data. When we let old data decay (see Figure 13), twenty eigensignals is enough to bring the energy fraction to nearly one; for the larger systems, this means we are compressing by orders of magnitude with minimal information loss.

C. Behavioral Subsystems

In this section, we discuss some practical applications of the output of the first stage of our analysis: the *behavioral subsystems*. An eigensignal describes the behavior of a subsystem over time; the weights of the subsystem capture how much each original signal contributes to the subsystem. Components may interact with each other to varying degrees, and our notion of a subsystem reflects this fact.

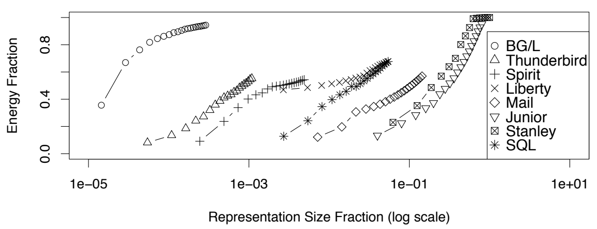

Fig. 12: The fraction of energy captured by the first 20 eigensignals, plotted versus the size of those signals as a fraction of the total input data. (Note that Stanley only has 16 components and therefore only 16 eigensignals.)

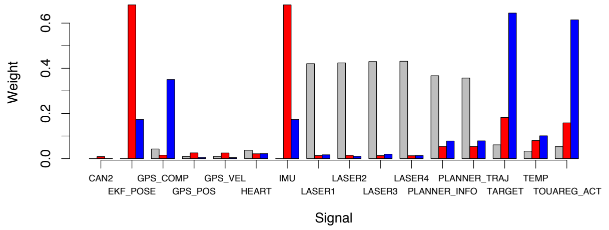

Fig. 14: Weights for Stanley's first three subsystems. The left bar indicates the absolute weight of that signal's contribution to the subsystem; the second bar indicates its weight in the second subsystem, etc.

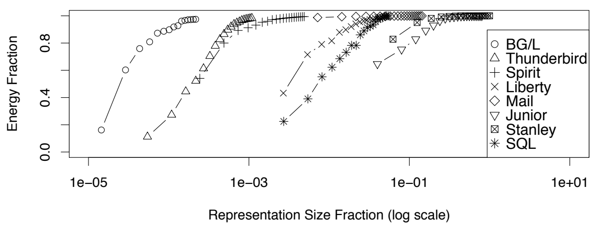

Fig. 13: When old data is allowed to be forgotten (decay), the behavior of the system can be described efficiently using a small number of eigensignals.

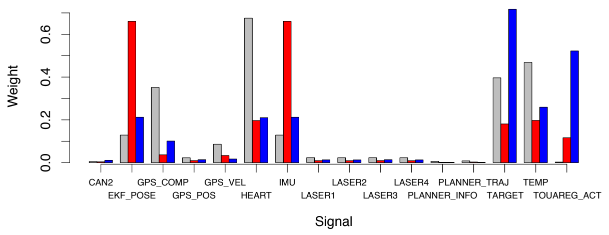

Fig. 15: Weights of Stanley's first three subsystems, with decay. The subsystem involving the lasers (see Figure 14) has long since decayed because the relevant anomalies happened early in the race.

1) Identifying Subsystems: During the Grand Challenge race, Stanley experienced a critical bug that caused the vehicle to swerve around nonexistent obstacles [38]. The Stanford Racing Team eventually learned that the laser sensors were sometimes misbehaving, but our analysis reveals a surprising interaction: the first subsystem is dominated by the laser sensors and the planner software (see Figure 14). This interaction was surprising because there was initially no apparent reason why four physically separate laser sensors should experience anomalies around the same time; it was also interesting that the planner software was correlated with these anomalies moreso than with the other sensors. As it turned out, there was an uninstrumented, shared component of the lasers that was causing this correlated behavior [25], [27] and whose existence our method was able to infer. This insight was critical to understanding the bug.

Administrators often ask, "What changed?" For example, does the interaction between Stanley's lasers and planner software persist throughout the log, or is it transient? The output of our analysis in Figure 15, which only reflects behavior near the end of the log, shows that the subsystem is transient. Most of the anomalies in the lasers and planner software occurred near the beginning of the race and are long-since forgotten by the end. As a result, the first subsystem is instead described by signals like the heartbeat and temperature sensor (which was especially anomalous near the end of the race because of the increasing desert heat). We currently identify temporal changes manually, but we could automate the process by comparing the composition of subsystems identified by the signal compression stage.

Subsystems can describe global behavior as well as local behavior. Figure 16 shows the weights for Spirit's first subsystem, whose representative is the aggregate signal of all the compute nodes; this subsystem describes a system-wide phenomenon (nodes exhibit more interesting behavior when they are running jobs). This is an example of behavior an administrator might choose to filter out of the anomaly signals. Meanwhile, the weights for Spirit's third subsystem, shown in Figure 17, are concentrated in a catch-all logging signal, signals related to component `sn111`, and alert types `R_HDA_NR` and `R_HDA_STAT` (which are hard drive-related problems [26]). This subsystem conveniently describes a specific kind of problem affecting a specific component, and knowing that those two types of alerts tend to happen together can help narrow down the list of potential causes.

2) Refining Instrumentation: Subsystem weights elucidate the extent to which sets of signals are redundant and which signals contain valuable information. There is operational value in refining the set of signals to include only those that give new information. The administrator of our SQL cluster stated this need as follows: "One of the problems with developing a set of metrics to measure how well a particular service is doing is that it's very easy to come up with an overwhelming number of them. However, if one wants to manage a service to metrics, one wants to have a reasonably small number of metrics to look at."

In addition to identifying redundant signals, subsystems can draw attention to places where more instrumentation would be helpful. After our analysis of the SQL cluster revealed that slow queries were predictive of bad downstream behavior, the administrator said, "I wish I had connection logs from other possible query sources to the MySQL servers to see if any of those would have uncovered a correlation [but] we don't save those in a useful fashion. This is pointing to some real deficiencies in our MySQL logging."

3) Representatives: When diagnosing problems in large systems, it is helpful to be able to decompose the system into pieces. Administrators currently do this using topological

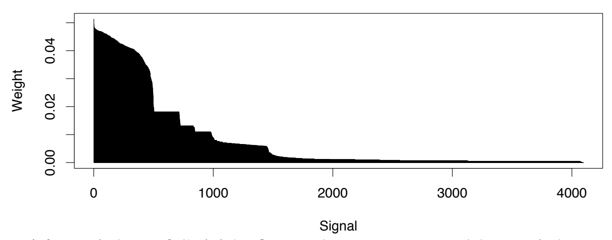

Fig. 16: Weights of Spirit's first subsystem, sorted by weight magnitude. The compression stage has identified a phenomenon that affects many of the components.

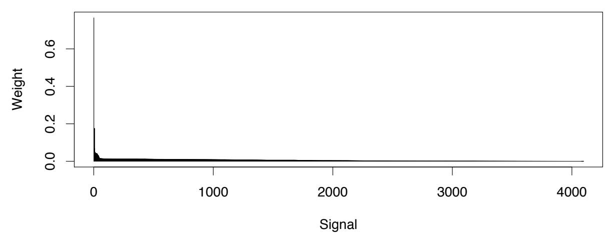

Fig. 17: Sorted weights of Spirit's third subsystem. Most of the weight is in a small subset of the components.

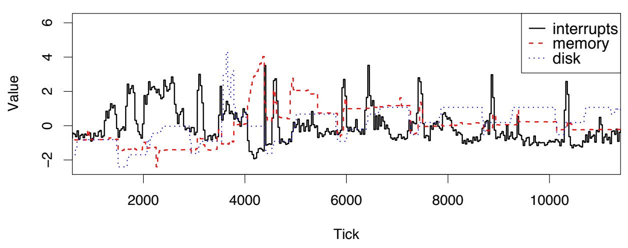

Fig. 18: The anomaly signals of the representatives of the first three subsystems for the SQL cluster.

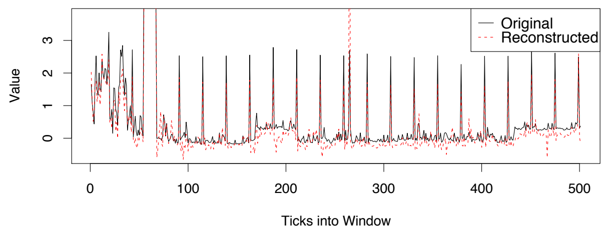

Fig. 19: Reconstruction of a portion of Liberty's `admin` signal using the subsystems, including the periodic anomalies.

information (e.g., is the problem more likely to be in Rack 1 or Rack 2?). Our analysis shows that topology is often a reasonable proxy for behavioral groupings. The representative signal for the first subsystem of many of the systems are aggregate signals: the aggregate signal summarizing interrupts in the SQL cluster, the `mail`-format logs from Mail cluster, the set of compute nodes in Liberty and Spirit, the components in Rack D of Thunderbird, and Rack 35 of BG/L. On the other hand, our experiments also revealed a variety of subsystems for which the representative signals were not topologically related. In other words, topological proximity does not imply correlated behavior nor does correlation imply topological proximity. For example, based on Figure 14, an administrator for Stanley would know to think about the laser sensors and planner software, together, as a subsystem.

A representative signal is also useful for quickly understanding what behaviors a subsystem describes. Figure 18 shows the anomaly signals of the representatives of the SQL cluster's first three subsystems. Based on the representatives, we can infer that these subsystems correspond to interrupts, application memory usage, and disk usage, respectively, and that these subsystems are not strongly correlated.

4) Collective failures: Behavioral subsystems can describe collective failures. On Thunderbird, there was a known system message suggesting a CPU problem: "`kernel: Losing some ticks... checking if CPU frequency changed.`" Among the signals generated for Thunderbird were signals that indicate when individual components output the message above. It turns out that this problem had nothing to do with the CPU; in fact, an operating system bug was causing the kernel to miss interrupts during heavy network activity. As a result, these messages were typically generated around the same time on multiple different components. Our method automatically notices this behavior and places these indicator signals into a subsystem: all of the first several hundred most strongly-weighted signals in Thunderbird's third subsystem were indicator signals for this "CPU" message. Knowing about this spatial correlation would have allowed administrators to diagnose the bug more quickly [25].

5) Missing Values and Reconstruction: Our analysis can deal gracefully with missing data because it explicitly guesses at the values it will observe during the current tick before observing them and adjusting the subsystem weights (see Section III-B). If a value is missing, the guessed value may be used, instead.

We can also output a reconstruction of the original anomaly signals using only the information in the subsystems (i.e., the weights and the eigensignals), meaning an administrator can answer historical questions about what the system was doing around a particular time, without the need to explicitly archive all the historical anomaly signals (which doesn't scale). Figure 19 shows the reconstruction of a portion of Liberty's `admin` anomaly signal. Most of this behavior is captured by the first subsystem, for which `admin` is representative.

Allowing older values to decay permits faster tracking of new behavior at the expense of seeing long-term trends. Figure 20 shows the reconstruction of one of Liberty's indicator signals, with decay. The improvement in reconstruction accuracy when using decay is apparent from Figure 21, which shows the relative reconstruction error for the SQL cluster. The behavior of this cluster changed near the end of the log as a result of an upgrade; the analysis with decay adapts to this change more easily.

D. Delays, Skews, and Cascades

In real systems, interactions may occur with some delay (e.g., high latency on one node eventually causes traffic to be rerouted to a second node, which causes higher latency on that second node a few minutes later) and may involve subsystems. We call these interactions *cascades*.

1) Cascades: The logs were rich with instances of individual signals and behavioral subsystems with lag correlations.

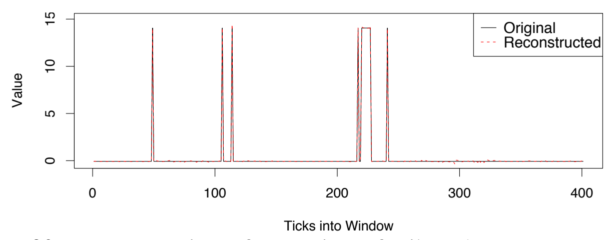

Fig. 20: Reconstruction of a portion of Liberty's `R_EXT_CCISS` indicator signal with decay.

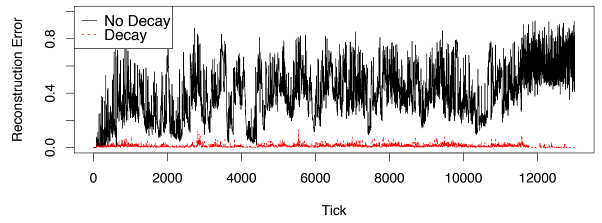

Fig. 21: Relative reconstruction error for the SQL cluster, with and without decay. Reconstruction is more accurate when old values decay, especially during a new phase near the end of this log.

This includes the supercomputer logs, whose anomaly signals have 1-hour granularity. We give a couple of examples here.

We first describe a cascade in Stanley: the critical swerving bug mentioned in Section V-C1, which has previously been analyzed only offline. Recall that the first stage of our analysis identifies one transient subsystem whose top four components are the four laser sensors and another subsystem whose top three components are the two planner components and the heartbeat component. The second stage discovers a lag correlation between these two subsystems with magnitude 0.47 and lag of 111 ticks (4.44 seconds). This agrees with the lag correlation between individual signals within the corresponding subsystems; e.g., `LASER4` and `PLANNER_TRAJ` have a maximum correlation magnitude of 0.65 at a lag of 101 ticks.

In Section I, we described a cascade using three real signals called `disk`, `forks`, and `swap`. These three signals (renamed for conciseness) are from the SQL cluster and are the top two components of the third subsystem and the representative of the fourth subsystem, respectively. Our method reports a lag correlation between the third and fourth subsystems of 30 minutes (see Figure 22). The administrator had been trying to understand this cascading behavior for weeks; our analysis confirmed one of his theories and suggested several interactions of which he had been unaware.

The administrator of the SQL cluster ultimately concluded that there was not enough information in the logs to definitively diagnose the underlying mechanism at fault for the crashes. This is a limitation of the data, not the analysis. In fact, in this example, our method both identified the shortcoming in the logs (a future logging change is planned as a result) and, despite the missing data, pointed at a diagnosis.

2) Online Alarms: Knowledge of a cascade may be actionable even as the cascade is underway and even when we do not understand the underlying cause. For instance, we can set alarms to trigger when the first sign of a cascade is detected. In the case of Stanley's swerving bug cascade, the Racing Team tells us Stanley could have prevented the swerving behavior by simply stopping whenever the lasers started to misbehave.

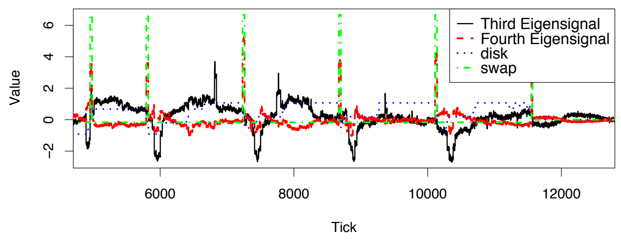

Fig. 22: In the SQL cluster, the strongest lag correlation was found between the third and fourth subsystems, with a magnitude of 0.46 and delay of 30 minutes. These eigensignals and their representatives' signals (`disk` and `swap`, respectively), are shown above.

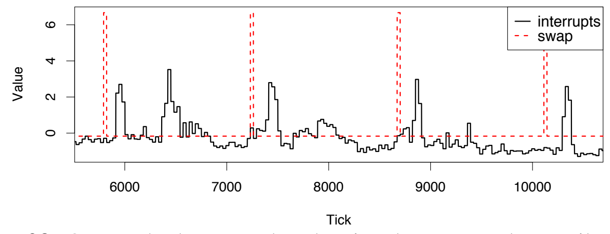

Fig. 23: Our method reports that the signal `swap` tends to spike 210 minutes before `interrupts`, with a correlation of 0.271; we can detect this online.

Some cascades operate on timescales that would allow more elaborate reactions or even human intervention. We tried the following experiment based on two of the lag-correlated signals reported by our method (plotted in Figure 23 and discussed briefly in Section I): when `swap` rises above a threshold, we raise an alarm and see how long it takes before we see `interrupts` rise above the same threshold. We use the first half of the log to determine and set the threshold to one standard deviation from the mean; we use the second half for our experiments, which yield no false positives and raise three alarms with an average warning time of 190 minutes. Setting the threshold at two standard deviations gives identical results. Depending on the situation, advanced warning about these spikes could allow remedial action like migrating computation, adjusting resource provisions, and so on.

3) Clock Skews: A cascade discovered between signals or subsystems that are known to act in unison may be attributable to clock skew. Without this external knowledge of what should happen simultaneously, there is no way to distinguish a clock skew from a cascade based on the data; our analysis can determine that there is some lag correlation, not the cause of the lag. If the user sees a lag that is likely to be a clock skew, our analysis provides the amount and direction of that skew, as well as the affected signals.

Although there were no known instances of clock skew in our data sets, we experimented with artificially skewing the timestamps of signals known to be correlated. We tested a variety of signals from different systems with correlation strengths varying from 0.264 to 0.999, skewing them from between 1 and 25 ticks. The amount of skew computed by our online method never differed from the actual skew by more than a couple of ticks; in almost all cases, the error was zero.

E. Results Summary

Our results show that signal compression drastically increases the scalability of lag correlation (see Section V-A) and that this compression process identifies behavioral subsystems with minimal information loss (see Section V-B). Experiments on large production systems (see Sections V-C–V-D) reveal that our method can produce operationally valuable results under common conditions where other methods cannot be applied: noisy, incomplete, and heterogeneous logs generated by systems that we cannot modify or perturb and for which we have neither source code nor correctness specifications.

VI. Contributions

We present an efficient, two-stage, online method for discovering interactions among components and groups of components, including time-delayed effects, in large production systems. The first stage compresses a set of anomaly signals using a principal component analysis and passes the resulting eigensignals and a small set of other signals to the second stage, a lag correlation detector, which identifies time-delayed correlations. We show, with real use cases from eight unmodified production systems, that understanding behavioral subsystems, correlated signals, and delays can be valuable for a variety of system administration tasks: identifying redundant or informative signals, discovering collective and cascading failures, reconstructing incomplete or missing data, computing clock skews, and setting early-warning alarms.

Acknowledgments

The authors would like to thank the following people for their time, expertise, and data: Russ Allbery, Kevin Hall, Jon Stearley, Mike Montemerlo and the rest of the Stanford Racing Team, Yasushi Sakurai, Christos Faloutsos, and our anonymous reviewers.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, 2003.

[2] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *SIGCOMM*, 2007.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *OSDI*, 2004.

[4] M. Brodie, I. Rish, and S. Ma. Optimizing probe selection for fault localization. In *Workshop on Distributed Systems: Operations and Management (DSOM)*, 2001.

[5] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *IEEE IM*, 2001.

[6] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *NSDI*, 2004.

[7] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: problem determination in large, dynamic internet services. In *DSN*, June 2002.

[8] S. Chutani and H. Nussbaumer. On the distributed fault diagnosis of computer networks. In *IEEE Symposium on Computers and Communications*, 1995.

[9] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP*, 2005.

[10] C. Ensel. New approach for automated generation of service dependency models. In *Latin American Network Operation and Management Symposium (LANOMS)*, 2001.

[11] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *IEEE Symposium on Security and Privacy*, 2003.

[12] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A pervasive network tracing framework. In *NSDI*, 2007.

[13] D. Gao, M. K. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In *CCS*, pages 318–329, 2004.

[14] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *USENIX Technical*, 2006.

[15] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *USENIX Security*, pages 61–79, 2002.

[16] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *SOSP*, 2009.

[17] I. T. Jolliffe. *Principal Component Analysis*. Springer, 2002.

[18] S. Kandula, D. Katabi, and J.-P. Vasseur. Shrink: A tool for failure diagnosis in IP networks. In *MineNet Workshop at SIGCOMM*, 2005.

[19] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. IP fault localization via risk modeling. In *NSDI*, 2005.

[20] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. *LNCS*, 2003.

[21] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3S: debugging deployed distributed systems. In *NSDI*, 2008.

[22] X. Liu, W. Lin, A. Pan, and Z. Zhang. WiDS Checker: Combating bugs in distributed systems. In *NSDI*, 2007.

[23] J. C. Mogul. Emergent (mis)behavior vs. complex software systems. In *EuroSys*, 2006.

[24] M. Montemerlo et al. Junior: The Stanford entry in the Urban Challenge. *Journal of Field Robotics*, 2008.

[25] A. J. Oliner and A. Aiken. A query language for understanding component interactions in production systems. In *ICS*, 2010.

[26] A. J. Oliner, A. Aiken, and J. Stearley. Alert detection in system logs. In *ICDM*, December 2008.

[27] A. J. Oliner, A. V. Kulkarni, and A. Aiken. Using correlated surprise to infer shared influence. In *DSN*, 2010.

[28] A. J. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *DSN*, 2007.

[29] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Ganesha: Black-box fault diagnosis for MapReduce systems. Technical report, CMU-PDL-08-112, 2008.

[30] S. Papadimitriou, J. Sun, and C. Faloutsos. Streaming pattern discovery in multiple time-series. In *VLDB*, 2005.

[31] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, 2006.

[32] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. WAP5: black-box performance debugging for wide-area systems. In *WWW*, 2006.

[33] I. Rish, M. Brodie, N. Odintsova, S. Ma, and G. Grabarnik. Real-time problem determination in distributed systems using active probing. In *NOMS*, 2004.

[34] Y. Sakurai, S. Papadimitriou, and C. Faloutsos. BRAID: Stream mining through group lag correlations. In *SIGMOD*, 2005.

[35] R. Schwarz and F. Mettern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing*, 1994.

[36] A. Singh, P. Maniatis, T. Roscoe, and P. Druschel. Using queries for distributed monitoring and forensics. In *EuroSys*, 2006.

[37] The Computer Failure Data Repository (CFDR). The HPC4 data. `http://cfdr.usenix.org/data.html`, 2009.

[38] S. Thrun and M. Montemerlo, et al. Stanley: The robot that won the DARPA Grand Challenge. *Journal of Field Robotics*, 2006.

[39] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Online system problem detection by mining patterns of console logs. In *ICDM*, 2009.

[40] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP*, 2009.

[41] E. S. K. Yu and J. Mylopoulos. Understanding "why" in software process modelling, analysis, and design. In *ICSE*, Sorrento, Italy, May 1994.

[42] Y. Zhao, Z. Zhu, Y. Chen, D. Pei, and J. Wang. Towards efficient large-scale VPN monitoring and diagnosis under operational constraints. In *INFOCOM*, 2009.