# USING INFLUENCE TO UNDERSTAND COMPLEX SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Adam Jamison Oliner
September 2011

# Abstract

This thesis is concerned with understanding the behavior of complex systems, particularly in the common case where instrumentation data is noisy or incomplete. We begin with an empirical study of logs from production systems, which characterizes the content of those logs and the challenges associated with analyzing them automatically, and present an algorithm for identifying surprising messages in such logs.

The principal contribution is a method, called *influence*, that identifies relationships among components—even when the underlying mechanism of interaction is unknown—by looking for correlated surprise. Two components are said to share an influence if they tend to exhibit surprising behavior that is correlated in time. We represent the behavior of components as surprise (deviation from typical or expected behavior) over time and use signal-processing techniques to find correlations. The method makes few assumptions about the underlying systems or the data they generate, so it is applicable to a variety of unmodified production systems, including supercomputers, clusters, and autonomous vehicles.

We then extend the idea of influence by presenting a query language and online implementation, which allow the method to scale to systems with hundreds of thousands of signals. In collaboration with system administrators, we applied these tools to real systems and discovered correlated problems, failure cascades, skewed clocks, and performance bugs. According to the administrators, it also generated information useful for diagnosing and fixing these issues.

# Acknowledgements

It would be impossible to provide a complete list of the people and experiences that shaped this thesis, but what follows is my best attempt to express some semblance of attribution and gratitude to those who made this document what it is and made me who I am.

My first research experience was with Boris Katz at the MIT InfoLab, where I worked on parsing tools for the START project under the supervision of Jimmy Lin. This was my foot in the door, so to speak, and I am grateful to Boris for giving a bright-eyed but fundamentally unqualified undergraduate the opportunity to participate in world-class research.

As part of the MIT VI-A program, a Masters that involves a series of industry internships and a research thesis, I spent three summers and a semester at IBM Research in Yorktown Heights. Under the supervision of Ramendra Sahoo and the management of José Moreira and Manish Gupta, I found my own research passions. Together, we published work on job scheduling, checkpointing, quality of service guarantees, and critical event prediction. My group at IBM (the Blue Gene/L System Software team) struck a truly remarkable balance between generating new research contributions and building a system that took and then held, for an unprecedented amount of time, the #1 spot on the Top 500 Supercomputers list.

Many people at MIT helped form me into the kind of scientist I am today. Larry Rudolph taught me how to do academic research while supervising my VI-A Program Masters thesis. Many of our conversations contained nuggets of wisdom that guided my later work. Over the years, he has been a steadfast advisor and a friend. Patrick Henry Winston taught me how to communicate my ideas more effectively. Martin

Rinard, standing in for my undergraduate advisor, helped me choose the path of academic research and checked in on me regularly during this journey.

The first four years of my Ph.D. were supported by a U.S. Department of Energy High Performance Computer Science Fellowship. This facilitated the beginning of a relationship with Sandia National Labs that made much of this thesis work possible. I spent the summer of 2006 at the Lab in Albuquerque, supervised by Jon Stearley and managed by Jim Ang. The Nodeinfo algorithm (see Chapter 3) is based on a method Jon used in his Sisyphus tool, and the work on that algorithm and the study of supercomputer system logs (see Chapter 2) were both done in close collaboration.

There were numerous additional collaborations that are not represented as explicitly in this thesis, but which nonetheless contributed to it. Some of these people I wish to thank are Elizabeth Stinson, Patrick Lincoln, Steve Dawson, Linda Briesemeister, Jim Thornton, John Mitchell, Peter Kwan and the rest of the VERNIER Team; Kaustubh Joshi, Matti Hiltunen, and Rick Schlichting of AT&T Research; and Jim Larus and Moises Goldszmidt of MSR.

My work on understanding complex systems would not have been possible without access to production systems, especially to the instrumentation data they generate. I am indebted to a staggering number of system managers and administrators who provided access to and interpretation of such data. These people include Sue Kelly, Bob Ballance, Sophia Corwell, Ruth Klundt, Dick Dimock, Michael Davis, Jason Repik, Victor Kuhns, Matt Bohnsack, Jerry Smith, Randall LaViolette, and Josh England of SNL; Kim Cupps, Adam Bertsch, Mike Miller, and Greg Bronevetsky of LLNL; John Daly of LANL; Russ Allbery and Kevin Hall of Stanford IT; Sebastian Gutierrez and Miles Davis of Stanford CS IT; Jonathan Mooser and Peter Kravtsov of Facebook; and Mike Montemerlo, Sebastian Thrun, and the rest of the Stanford Racing Team.

I was fortunate to work with four excellent Masters students during my time at Stanford: Naeim Semsarilar, Ashutosh Kulkarni, Anton Vogt, and James Fosco. Their hard work and keen insights shrunk implementation times from months to weeks.

I would also like to thank the members of my thesis committee: John Ousterhout, Dawson Engler, Subasish Mitra, and Lera Boroditsky.

Many of my collaborators also became good friends, or vice versa. Thank you Kathleen Fisher, Daniel Ramage, Peter Hawkins, Eric Schkufza, Paul Heymann, Philip Guo, Mike Bauer, and Xuân Vũ.

I am immensely thankful to my family and friends for their years of dedicated support. I love you all.

Finally, I want to express deep gratitude toward my advisor, Alex Aiken. The opportunity to work with Alex was one of the main reasons I chose to come to Stanford, and the years we worked together proved this to be the correct decision. I was a systems researcher in a programming languages group, truly a black sheep. This gave me an opportunity to see problems from a new perspective and consider solutions that might not have occurred to someone in a more typical position.

Thank you, everyone. Thank you.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Complex systems are pervasive: data centers drive our economy, telecommunication networks support our emergency services, and robots build many of the products we buy. Our dependence on such systems is increasing, and so is their complexity. There is a pressing need to gain insight into the behavior of such systems so we can diagnose misbehavior, fix bugs, optimize performance, and build better systems. This thesis focuses on understanding complex systems, particularly in the real-world case of large production systems where instrumentation data is noisy and incomplete.

Consider the computing infrastructure at a company like Facebook. Each data center consists of thousands of networked machines and each machine, in turn, has subsystems for memory, communication, and computation. This system receives input from and generates output for hundreds of millions of users and is always changing due to failures, upgrades, and other modifications. Misbehavior is often caused by complex component interactions. For example, recently an error checker for a configuration-value cache brought the entire site down for more than two hours[1]. These kinds of interactions occur even when, as is the case with Facebook, the system is constructed using current best software practices. Some components are instrumented to generate metrics (e.g., the throughput over a particular link) or console messages (e.g., "file settings.conf appears to be corrupt"); some components are proprietary or critical to performance and therefore generate no such data. These logs

[1] `http://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919`

are the primary source of insight into the behavior of production systems, yet there has been relatively little research on how to effectively use the raw data these systems generate, primarily due to the difficulty of obtaining such data.

This thesis begins, in Chapter 2, with a study of raw logs from multiple supercomputers [46]: Blue Gene/L, Thunderbird, Red Storm, Spirit, and Liberty. We present details about the systems, methods of log collection, and how important messages were identified; propose a simpler and more effective filtering algorithm; and define *operational context* to encode crucial information that was found to be missing from most logs. The chapter explains the practical challenges of log analysis and suggests promising research directions for work in data mining, filtering, root cause analysis, and critical event prediction. System logs are the first place system administrators go when they are alerted to a problem, and are one of the few resources available to them for gaining visibility into the behavior of a machine. Studying these logs was a crucial first step toward understanding the behavior of complex systems.

Building on these results, Chapter 3 presents an unsupervised algorithm called Nodeinfo for identifying important messages called *alerts* in system logs [38], which we use to create the first verifiable and reproducible performance baseline for alert detection. Using data from four production supercomputers, in which alerts were manually tagged for testing, the results demonstrate Nodeinfo's effectiveness. That chapter formalizes the alert detection task, describes how Nodeinfo uses the information entropy of message terms to identify alerts, and presents an online version of this algorithm that is now in production use on at least three supercomputers. In addition to finding known alerts more accurately than existing methods, Nodeinfo also discovered several kinds of alerts that were previously unknown to the administrators and that enabled them to take remedial action.

Chapter 4 proposes a method for identifying the sources of problems in complex production systems where the data available for analysis may be noisy or incomplete (recall the Facebook example). The results on systems like supercomputers, clusters, and autonomous vehicles show that our approach identifies important relationships among components, even for very large systems, and that this information is useful to system administrators. We define *influences* as a class of component interactions that

includes direct communication and resource contention [39]. Intuitively, the method infers influences by looking for correlated surprise; we represent the behavior of each component as surprise-over-time and use signal-processing techniques to find pairs of components with time-correlated anomalous behavior. (One source of such surprise signals is the alert detection results of Chapter 3.) For example, a hard disk might regularly generate a strange log message seconds before an unusual increase in request latency, leading the method to infer that the disk has a (likely causal) influence on responsiveness.

To facilitate the process of computing influence, Chapter 5 introduces a query language and a method for computing queries that makes few assumptions about the available data [44]. In collaboration with the system administrators, we tested this method of querying influence on billions of lines of logs from unmodified systems. Some of these systems (e.g., Blue Gene/L) had hundreds of thousands of signals. The tool discovered correlated problems, failure cascades, skewed clocks, and performance bugs. According to the administrators, it also generated information useful for diagnosing and fixing these issues.

Finally, Chapter 6 extends the idea of computing influence by making the analysis online and more scalable [45]. Using recent results in algorithms for principal components analysis and lag correlation detection, we implemented a real-time method for computing influence in production systems. The process scales to systems with hundreds of thousands of components and, by virtue of being an online algorithm, empowers an administrator to perform tasks that would be impossible in an offline setting. For example, in one experiment, our analysis detected a failure cascade in a database cluster, with a duration of several hours, that culminated in a machine crash. After detecting this cascade pattern, our tool was able to set an alarm for the second half of the log that would have warned the administrator more than three hours in advance of the crashes, with no false positives or false negatives.

The primary lessons of this thesis are as follows:

- Instrumentation in large systems is noisy and incomplete, but does contain useful information.

- Similar components running similar workloads tend to generate similar logs. Messages that deviate from this regularity tend to be subjectively interesting to system administrators and often indicate misbehavior.
- It is possible to glean an understanding of high-level component and subsystem interactions using such low-level measurements. These interactions can be crucial clues for diagnosing misbehavior.
- Correlation can be a powerful diagnosis tool, especially in the common case where the data is insufficient to imply causality. Our approach correlates surprise (deviation from expectation or regularity) rather than the noisy, raw measurements.
- Representing behavior with real-valued signals, rather than discretized categorical data, provides greater power to detect time-delayed effects and subtle variations.

We support these observations with data from a variety of production systems and with experiments that involved close collaboration with the system administrators. The results show that our influence method can produce valuable results even under conditions where alternative methods are inapplicable.

# Chapter 2

# System Logs

The reliability and performance challenges of large, complex systems cannot be adequately addressed until the behavior of the machines is better understood. Progress has been hampered by the inaccessibility of data from such production systems. In this chapter, we study system logs from five of the world's most powerful supercomputers. The machines we consider (and the number of processors) are: Blue Gene/L (131072), Red Storm (10880), Thunderbird (9024), Spirit (1028), and Liberty (512). The analysis encompasses more than 111.67 GB of data containing 178,081,459 alert messages in 77 categories. The system logs are the first place system administrators go when they are alerted to a problem, and are one of the few mechanisms available to them for gaining visibility into the behavior of the machine. Particularly as systems grow in size and complexity, there is a pressing need for better techniques for processing, understanding, and applying these data.

We define an *alert* to be a message in the system logs that merits the attention of the system administrator, either because immediate action must be taken or because there is an indication of an underlying problem. Many alerts may be symptomatic of the same *failure*. Failures may be anything from a major filesystem malfunction to a transient connection loss that kills a job.

Using results from the analysis, we give lessons learned for future research in this area. Most importantly, we discuss the following issues:

- Logs do not currently contain sufficient information to do automatic detection

of failures, nor root cause diagnosis, with acceptable confidence. Although identifying candidate alerts is tractable, disambiguation in many cases requires external context that is not available. The most salient missing data is *operational context*, which captures the system's expected behavior.

- There is a chaotic effect in these systems, where small events or changes can dramatically impact the logs. For instance, an OS upgrade on Liberty instantaneously increased the average message traffic. On Spirit, a single node experiencing disk failure produced the majority of all log messages.

- Different categories of failures have different predictive signatures (if any). Event prediction efforts should produce an ensemble of predictors, each specializing in one or more categories.

- Along with the issues above, automatic identification of alerts must deal with: corrupted messages, inconsistent message structure and log formats, asymmetric alert reporting, and the evolution of systems over time.

Section 2.1 describes the five supercomputers, the log collection paths, and the logs themselves. Section 2.1.2 explains the alert tagging process and notes what challenges will be faced by those hoping to do such tagging (or detection) automatically. Section 2.2 contains graphical and textual examples of the data and a discussion of the implications for filtering and modeling. Finally, Section 6.4 summarizes the contributions and lessons learned.

The purpose of this chapter is not to argue for a particular reliability, availability, and serviceability (RAS) architecture, nor to compare the reliability of the supercomputers. The systems we study are real, and the logs are in the form used by (and familiar to) system administrators. Our intention is to elucidate the practical challenges of log analysis for supercomputers, and to suggest fruitful research directions for work in data mining, filtering, root cause analysis, and critical event prediction. This is the first study, to our knowledge, that has considered raw logs from multiple supercomputing systems.

| System | Owner | Vendor | Rank | Procs | Memory (GB) | Interconnect |
|---|---|---|---|---|---|---|
| Blue Gene/L | LLNL | IBM | 1 | 131072 | 32768 | Custom |
| Thunderbird | SNL | Dell | 6 | 9024 | 27072 | Infiniband |
| Red Storm | SNL | Cray | 9 | 10880 | 32640 | Custom |
| Spirit (ICC2) | SNL | HP | 202 | 1028 | 1024 | GigEthernet |
| Liberty | SNL | HP | 445 | 512 | 944 | Myrinet |

Table 2.1: System characteristics at the time of collection. External system names are indicated in parentheses. Information such as Rank was obtained from the Top500 Supercomputer list [71]. The machines are representative of the design choices and scales seen in current supercomputers.

| System | Start Date | Days | Size (GB) | Compressed | Rate (bytes/sec) |
|---|---|---|---|---|---|
| Blue Gene/L | 2005-06-03 | 215 | 1.207 | 0.118 | 64.976 |
| Thunderbird | 2005-11-09 | 244 | 27.367 | 5.721 | 1298.146 |
| Red Storm | 2006-03-19 | 104 | 29.990 | 1.215 | 3337.562 |
| Spirit (ICC2) | 2005-01-01 | 558 | 30.289 | 1.678 | 628.257 |
| Liberty | 2004-12-12 | 315 | 22.820 | 0.622 | 835.824 |

Table 2.2: Log characteristics. Compression was done using the Unix utility `gzip`.

## 2.1 Supercomputer Logs

The broad range of supercomputers considered in this study are summarized in Table 2.1. All five systems are ranked on the Top500 Supercomputers List as of June 2006 [71], spanning a range from #1 to #445. They vary by two orders of magnitude in the number of processors and by one order of magnitude in the amount of main memory. The interconnects include Myrinet, Infiniband, GigEthernet, and custom or mixed solutions. The various machines are produced by IBM, Dell, Cray, and HP. All systems are installed at Sandia National Labs (SNL) in Albuquerque, NM, with the exception of BG/L, which is at Lawrence Livermore National Labs (LLNL) in Livermore, California.

### 2.1.1 Log Collection

It is standard practice to log messages and events in a supercomputing system; no special instrumentation nor monitoring was added for this study. Tables 2.2 and

| System | Messages | Alerts | Categories |
|---|---|---|---|
| Blue Gene/L | 4,747,963 | 348,460 | 41 |
| Thunderbird | 211,212,192 | 3,248,239 | 10 |
| Red Storm | 219,096,168 | 1,665,744 | 12 |
| Spirit (ICC2) | 272,298,969 | 172,816,564 | 8 |
| Liberty | 265,569,231 | 2,452 | 6 |

Table 2.3: Message breakdown. The number of alerts reflects redundant reporting and the preferences of the system administrators more than it indicates the reliability of the system. Alerts were tagged into categories according to the heuristics supplied by the administrators for the respective systems, as described in Section 2.1.2. Two alerts are in the same category if they were tagged by the same expert rule; the categories column indicates the number of categories that were actually observed in each log.

2.3 present an overview of the logs. The remainder of this section focuses on the infrastructure that generated them.

On Thunderbird, Spirit, and Liberty, logs are generated on each local machine by `syslog-ng` and both stored to `/var/log/` and sent to a logging server. The logging servers (`tbird-admin1` on Thunderbird, `sadmin2` on Spirit, and `ladmin2` on Liberty) process the files with `syslog-ng` and place them in a directory structure according to the source node. We collected the logs from that directory. As is standard syslog practice, the UDP protocol is used for transmission, resulting in some messages being lost during network contention.

Red Storm has several logging paths. Disk and RAID controller messages in the DDN subsystem pass through a 100 Megabit network to a DDN-specific RAS machine, where they are processed by `syslog-ng` and stored. Similarly, all Linux nodes (login, Lustre I/O, and management nodes) transmit syslog messages to a different `syslog-ng` collector node for storage. All other components (compute nodes, SeaStar NICs, and hierarchical management nodes) generate messages and events which are transmitted through the RAS network (using the reliable TCP protocol) to the System Management Workstation (SMW) for automated response and storage. Our study includes all of these logs.

On BG/L, logging is managed by the Machine Management Control System

| Type | Raw | | Filtered | |
|---|---|---|---|---|
| | Count | % | Count | % |
| Hardware | 174,586,516 | 98.04 | 1,999 | 18.78 |
| Software | 144,899 | 0.08 | 6,814 | 64.01 |
| Indeterminate | 3,350,044 | 1.88 | 1,832 | 17.21 |

Table 2.4: Hardware was the most common type of alert, but not the most common type of failure (as estimated by the filtered results). Filtering dramatically changes the distribution of alert types.

(MMCS), which runs on the service node, of which there are two per rack [1]. Compute chips store errors locally until they are polled, at which point the messages are collected via the JTAG-mailbox protocol. The polling frequency for our logs was set at around one millisecond. The service node MMCS process then relays the messages to a centralized DB2 database. That RAS database was the source of our data, and includes hardware and software errors at all levels, from chip SRAM parity errors to fan failures. Events in BG/L often set various RAS flags, which appear as separate lines in the log. The time granularity for BG/L logs is down to the microsecond, unlike the one-second granularity of typical syslogs. This study does not include syslogs from BG/L's Lustre I/O cluster and shared disk subsystem.

### 2.1.2 Identifying Alerts

For each of the systems, we worked in consultation with the respective system administrators to determine the subset of log entries that they would *tag* as being alerts. Thus, the alerts we identify in the logs are certainly alerts by our definition, but the set is (necessarily) not exhaustive. In all, we identified 178,081,459 alerts across the logs; see Table 2.2 for the breakdown by system and Table 2.1.2 for the alerts, themselves. Alerts were assigned *types* based on their ostensible subsystem of origin (hardware, software, or indeterminate); this is based on each administrator's best understanding of the alert, and may not necessarily be root cause. Table 2.4 presents the distribution of types both before and after filtering (described in Section 2.1.3).

Table 2.1.2 provides example alert messages from the supercomputers. System names are listed with the total number alerts before and after filtering. "Cat." is the

alert category. Types are H (Hardware), S (Software), and I (Indeterminate). Indeterminate alerts can originate from both hardware and software, or have unknown cause. Due to space, we list only the most common of the 41 BG/L alert categories. Bracketed text indicates information that is omitted; a bracketed ellipsis indicates sundry text. Alert categories vary among machines as a function of system configurations, logging mechanisms, and what each system's administrators deem important.

| Alert Type/Cat. | Raw | Filtered | Example Message Body (Anonymized) |
|---|---|---|---|
| **BG/L** | **348,460** | **1202** | |
| H / KERNDTLB | 152,734 | 37 | `data TLB error interrupt` |
| H / KERNSTOR | 63,491 | 8 | `data storage interrupt` |
| S / APPSEV | 49,651 | 138 | `ciod: Error reading message prefix after LOGIN_MESSAGE on CioStream` [...] |
| S / KERNMNTF | 31,531 | 105 | `Lustre mount FAILED : bglio11 : block_id : location` |
| S / KERNTERM | 23,338 | 99 | `rts: kernel terminated for reason 1004rts: bad message header:` [...] |
| S / KERNREC | 6145 | 9 | `Error receiving packet on tree network, expecting type 57 instead of` [...] |
| S / APPREAD | 5983 | 11 | `ciod: failed to read message prefix on control stream` [...] |
| S / KERNRTSP | 3983 | 260 | `rts panic! - stopping execution` |
| S / APPRES | 2370 | 13 | `ciod: Error reading message prefix after LOAD_MESSAGE on CioStream` [...] |
| I / APPUNAV | 2048 | 3 | `ciod: Error creating node map from file` [...] |
| I / 31 Others | 7186 | 519 | `machine check interrupt` |
| **Thunderbird** | **3,248,239** | **2088** | |
| I / VAPI | 3,229,194 | 276 | `kernel: [KERNEL_IB]`[...] `(Fatal error (Local Catastrophic Error))` |
| S / PBS_CON | 5318 | 16 | `pbs_mom: Connection refused (111) in open_demux, open_demux: cannot` [...] |
| I / MPT | 4583 | 157 | `kernel: mptscsih: ioc0: attempting task abort! (sc=00000101bddee480)` |
| H / EXT_FS | 4022 | 778 | `kernel: EXT3-fs error (device sda5):` [...] `Detected aborted journal` |
| S / CPU | 2741 | 367 | `kernel: Losing some ticks... checking if CPU frequency changed.` |
| H / SCSI | 2186 | 317 | `kernel: scsi0 (0:0): rejecting I/O to offline device` |
| H / ECC | 146 | 143 | `Server Administrator: Instrumentation Service EventID: 1404 Memory device` [...] |

| | | | |
|---|---|---|---|
| S / PBS_BFD | 28 | 28 | pbs_mom: Bad file descriptor (9) in tm_request, job [job] not running |
| H / CHK_DSK | 13 | 2 | check-disks: [node:time] , Fault Status assert [...] |
| I / NMI | 8 | 4 | kernel: Uhhuh. NMI received. Dazed and confused, but trying to continue |
| **Red Storm** | **1,665,744** | **1430** | |
| H / BUS_PAR | 1,550,217 | 5 | DMT_HINT Warning: Verify Host 2 bus parity error: 0200 Tier:5 LUN:4 [...] |
| I / HBEAT | 94,784 | 266 | ec_heartbeat_stop\|src:::[node] \|svc:::[node]warn\|node heartbeat fault\|[...] |
| I / PTL_EXP | 11,047 | 421 | kernel: LustreError: [...] @@@ timeout (sent at [time], 300s ago) [...] |
| H / ADDR_ERR | 6763 | 1 | DMT_102 Address error LUN:0 command:28 address:f000000 length:1 Anonymous [...] |
| H / CMD_ABORT | 1686 | 497 | DMT_310 Command Aborted: SCSI cmd:2A LUN 2 DMT_310 Lane:3 T:299 a: [...] |
| I / PTL_ERR | 631 | 54 | kernel: LustreError: [...] @@@ type == [...] |
| I / TOAST | 186 | 9 | ec_console_log\|src:::[node]\|svc:::[node]\| PANIC_SP WE ARE TOASTED! |
| I / EW | 163 | 58 | kernel: Lustre:[...] Expired watchdog for pid[job] disabled after [#]s |
| I / WT | 107 | 45 | kernel: Lustre:[...] Watchdog triggered for pid[job]: it was inactive for [#]ms |
| I / RBB | 105 | 19 | kernel: LustreError: [...] All mds cray_kern_nal request buffers busy (0us idle) |
| H / DSK_FAIL | 54 | 54 | DMT_DINT Failing Disk 2A |
| I / OST | 1 | 1 | kernel: LustreError: [...] Failure to commit OST transaction (-5)? |
| **Spirit** | **172,816,564** | **4875** | |
| H / EXT_CCISS | 103,818,910 | 29 | kernel: cciss: cmd 0000010000a60000 has CHECK CONDITION, sense key = 0x3 |
| H / EXT_FS | 68,986,084 | 14 | kernel: EXT3-fs error (device[device]) in ext3_reserve_inode_write: IO failure |
| S / PBS_CHK | 8388 | 4119 | pbs_mom: task_check, cannot tm_reply to [job] task 1 |
| S / GM_LANAI | 1256 | 117 | kernel: GM: LANai is not running. Allowing port=0 open for debugging |
| S / PBS_CON | 817 | 25 | pbs_mom: Connection refused (111) in open_demux, open_demux: connect [IP:port] |
| S / GM_MAP | 596 | 180 | gm_mapper[[#]]: assertion failed. [path]/lx_mapper.c:2112 (m->root) |
| S / PBS_BFD | 346 | 296 | pbs_mom: Bad file descriptor (9) in tm_request, job [job] not running |

| | | | |
|---|---|---|---|
| H / GM_PAR | 166 | 95 | kernel: GM: The NIC ISR is reporting an SRAM parity error. |
| **Liberty** | **2452** | **1050** | |
| S / PBS_CHK | 2231 | 920 | pbs_mom: task_check, cannot tm_reply to [job] task 1 |
| S / PBS_BFD | 115 | 94 | pbs_mom: Bad file descriptor (9) in tm_request, job [job] not running |
| S / PBS_CON | 47 | 5 | pbs_mom: Connection refused (111) in open_demux, open_demux: connect [IP:port] |
| H / GM_PAR | 44 | 19 | kernel: GM: LANAI[0]: PANIC: [path]/gm_parity.c:115:parity__int():firmware |
| S / GM_LANAI | 13 | 10 | kernel: GM: LANai is not running. Allowing port=0 open for debugging |
| S / GM_MAP | 2 | 2 | gm_mapper[736]: assertion failed. [path]/mi.c:541 (r == GM_SUCCESS) |

Table 2.1.2: Alert breakdown and examples.

Note that many of these alerts were multiply reported by one or more nodes (sometimes millions of times), requiring filtering of the kind discussed in Section 2.1.3. Furthermore, it means that the number of alerts we report does not necessarily relate to the reliability of the systems in any meaningful way. The heuristics provided by the administrators were often in the form of regular expressions amenable for consumption by the `logsurfer` utility [49]. We performed the tagging through a combination of regular expression matching and manual intervention. The administrators with whom we consulted were responsible for their respective systems throughout the period of log collection and the publication of this work. Examples of alert-identifying rules using `awk` syntax include (from Spirit, Red Storm, and BG/L, respectively) include the following:

```
/kernel: EXT3-fs error/
/PANIC_SP WE ARE TOASTED!/
($5 ~ /KERNEL/ && /kernel panic/)
```

Previous work on BG/L log analysis used simple alert identification schemes such as the *severity* field of messages [27, 28, 55] or an external source of information [58, 68]. Because our objective was not to suggest an alert detection scheme, but rather to accurately characterize the content of the logs, we instead used the time-consuming

| Severity | Messages | | Alerts | |
|---|---|---|---|---|
| | Count | % | Count | % |
| FATAL | 855,501 | 18.02 | 348,398 | 99.98 |
| FAILURE | 1714 | 0.03 | 62 | 0.02 |
| SEVERE | 19,213 | 0.41 | 0 | 0 |
| ERROR | 112,355 | 2.37 | 0 | 0 |
| WARNING | 23,357 | 0.49 | 0 | 0 |
| INFO | 3,735,823 | 78.68 | 0 | 0 |

Table 2.6: The distribution of severity fields for BG/L among all messages and among our expert-tagged alerts. Tagging all FATAL/FAILURE severity messages as alerts would have yielded a 59% false positive rate.

manual process described above. We discovered, furthermore, that administrators for these machines do not use the severity field as the singular way to detect alerts, and that many systems (Thunderbird, Spirit, and Liberty) did not even record this information.

Table 2.6 shows the distribution of severity fields among messages and among unfiltered alerts. If we had used the severity field instead of the expert rules to tag alerts on BG/L, tagging any message with a severity of `FATAL` or `FAILURE` as an alert, we would have a false negative rate of 0% but a false positive rate of 59.34%. Of the Sandia systems, only Red Storm is configured to store the severity of syslog messages (the Red Storm TCP log path is not syslog and has no severity analog). Table 2.7 gives the severity distribution, which suggests that syslog severity is of dubious value as a failure indicator. The use of message severity levels as a criterion for identifying failures should be done only with considerable caution.

### Alert Identification Challenges

Automatically identifying alerts in system logs is an open problem. To facilitate others in tackling this challenge, we offer the following account of issues we observed while manually tagging the logs that must be addressed by an automated scheme:

**Insufficient Context.** Many log messages are ambiguous without external context. The most salient piece of missing information was what we call *operational context*,

| Severity | Messages | | Alerts | |
|---|---|---|---|---|
| | Count | % | Count | % |
| EMERG | 3 | 0.00 | 0 | 0 |
| ALERT | 654 | 0.00 | 45 | 0.00 |
| CRIT | 1,552,910 | 6.09 | 1,550,217 | 98.69 |
| ERR | 2,027,598 | 7.95 | 11,784 | 0.75 |
| WARNING | 2,154,944 | 8.45 | 270 | 0.02 |
| NOTICE | 3,759,620 | 14.74 | 0 | 0 |
| INFO | 15,722,695 | 61.63 | 8,450 | 0.54 |
| DEBUG | 291,764 | 1.14 | 0 | 0 |

Table 2.7: The distribution of severity fields for Red Storm syslogs among all messages and among our expert-tagged alerts. These syslog alerts were dominated by disk failure messages with CRIT severity. Except for this failure case, these data suggest that syslog severity is not a reliable failure indicator.

which helps to account for the human and other external factors that influence the semantics of log messages. For example, consider the following ambiguous example message from BG/L (anonymized):

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

This message has a very high severity (FAILURE), but the message body suggests that the program exited cleanly. If the system administrator were doing maintenance on the machine at the time, this message is a harmless artifact of his actions. On the other hand, if it were generated during normal machine operation, this message indicates that all running jobs on the supercomputer were (undesirably) killed. The disparity between these two interpretations is tremendous. Only with additional information supplied by the system administrator could we conclude that this message was likely innocuous. In our experience, operational context is one of the most vital, but often absent, factors in deciphering system logs.

As seen in Figure 2.1, operational context may indicate whether a system is in engineering or production time. Sandia, Los Alamos, and Livermore National Laboratories are currently working together to define exactly what information is needed, and how to use it to quantify RAS performance [63]. It may be sufficient to record only a few bytes of data: the time and cause of system state changes. For example, the

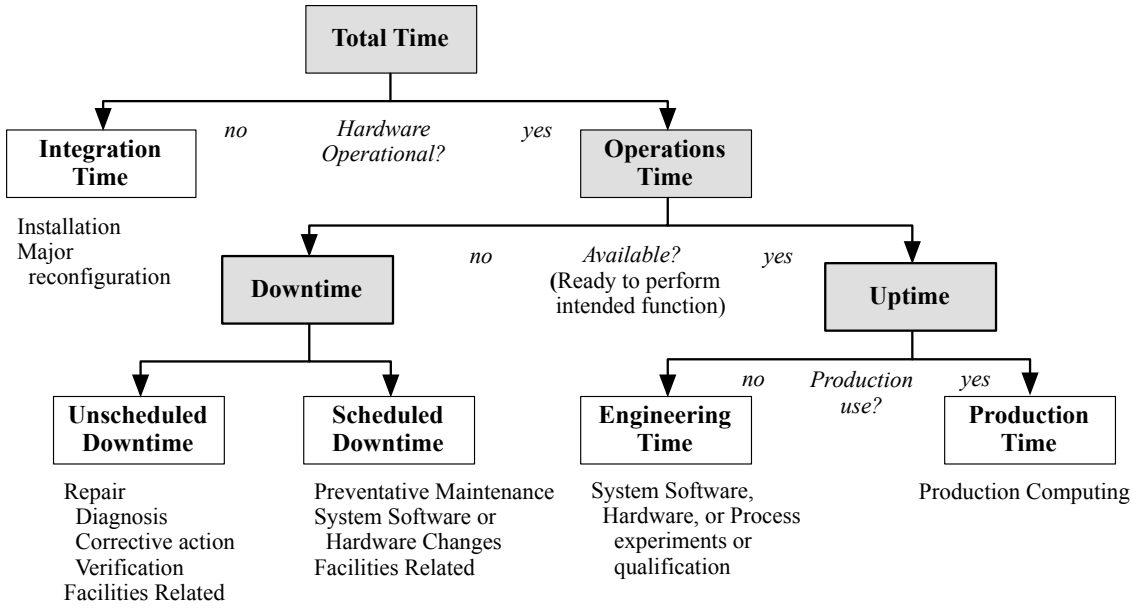



Figure 2.1: Operational context example. Event significance can be disambiguated if the expected state of components is known. This diagram is the current basis of Red Storm RAS metrics, and is being developed by LANL, LLNL, and SNL towards establishing standardized RAS performance metrics.

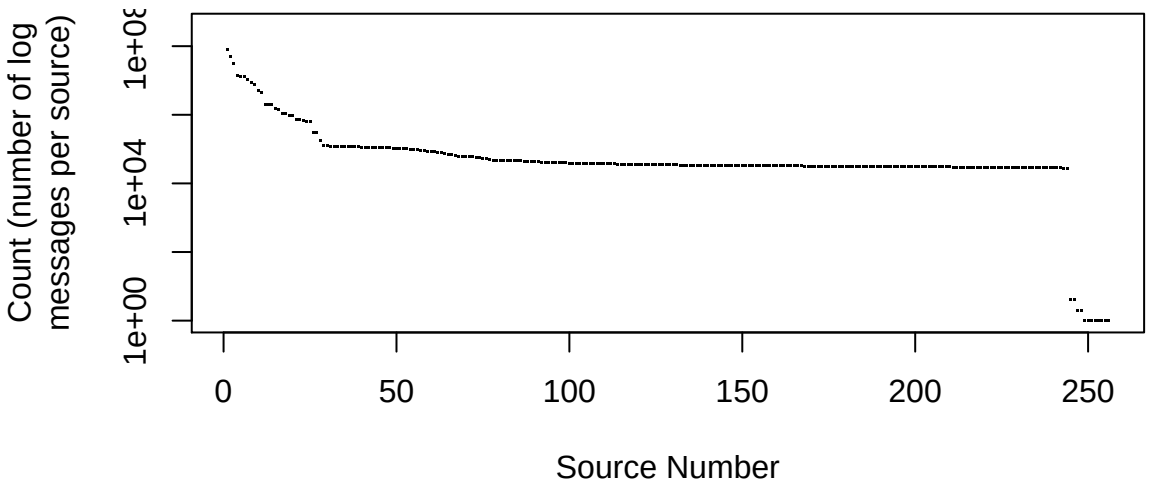



Figure 2.2: The number of sources (logical nodes) of messages in Liberty, sorted by decreasing quantity. The most prolific sources were administrative nodes or those with significant problems. The cluster on the bottom right is from the set of messages whose source field was corrupted, thwarting attribution.

commencement of an OS upgrade would be accompanied by a message indicating that at time $t$ the system entered *scheduled downtime* for a system software installation. A similar message would accompany the system's return to *production time*.

The lack of context has also affected the study of parallel workloads. Feitelson proposed removing non-production jobs from workload traces (such as workload flurries attributable to system testing [15]). Analogously, some alerts may be ignored during a scheduled downtime that would be significant during production time.

**Asymmetric Reporting.** Some failures leave no evidence in the logs, and the logs are fraught with messages that indicate nothing useful at all. More insidiously, even single failure types may produce varying alert signatures in the log. For example, the Red Storm DDN system generates a great variety of alert patterns that all mean "disk failure". Nodes also generate differing logs according to their function. Figure 2.2 shows the number of messages broken down by source. The chatty sources tended to be the administrative nodes or those with persistent problems, while the reticent sources were either misconfigured or improperly attributed (the result of corrupted messages).

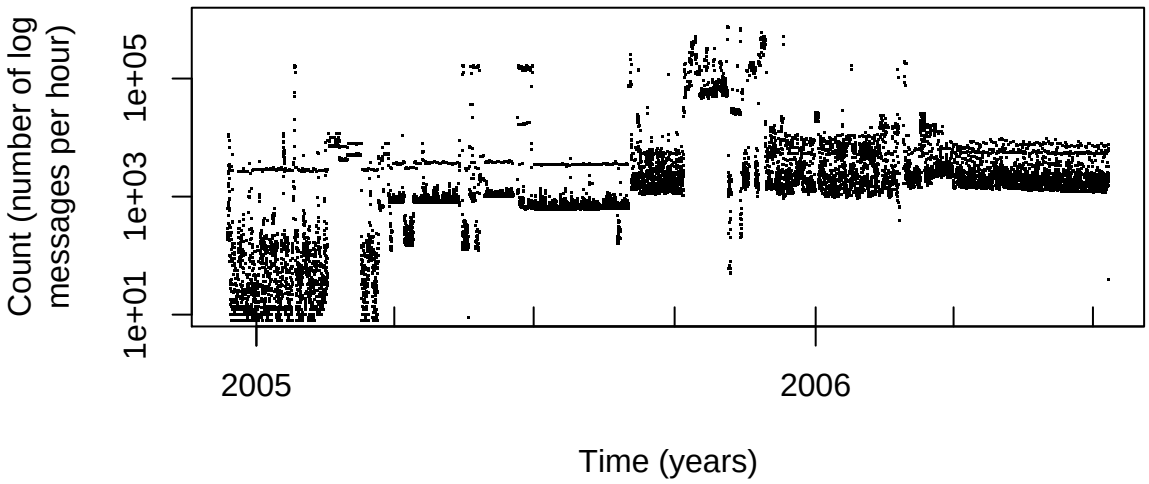



Figure 2.3: The number of messages generated by Liberty, bucketed by hour. The first major shift (end of first quarter, 2005) corresponds to an upgrade in the operating system after the machine was put into production use. The causes of the other shifts are not well understood at this time.

**System Evolution.** Log analysis is a moving target. Over the course of a system's lifetime, anything from software upgrades to minor configuration changes can drastically alter the meaning or character of the logs. Figure 2.3, for example, shows dramatic shifts in behavior over time. We have seen alerts that, while common in a prefix of the log, no longer appear after such shifts. This makes machine learning difficult: learned patterns and behaviors may not be applicable for very long. The ability to detect phase shifts in behavior would be a valuable tool for triggering relearning or for knowing which existing behavioral model to apply.

**Implicit Correlation.** Groups of messages are sometimes fundamentally related, but there is no explicit indication of this. See Figures 2.4 and 2.5. A common such correlation results from cascading failures.

**Inconsistent Structure.** Despite the noble efforts of the BSD syslog standard and others, log messages vary greatly both within and across systems. BG/L and Red Storm use custom databases and formats, and commodity syslog-based systems do not even record fields such as *severity* by default. Ultimately, understanding the entries

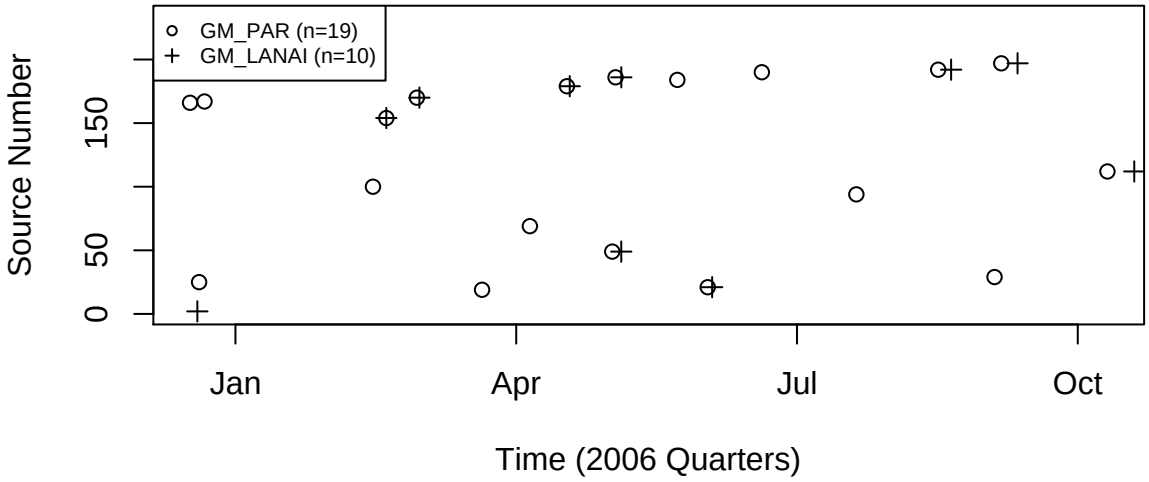



Figure 2.4: Two related classes of alerts from Liberty. Notice that GM_LANAI messages do not always follow GM_PAR messages, nor vice versa. However, the correlation is clear. Current tagging and filtering techniques do not adequately address this situation.

may require parsing the unstructured message bodies, thereby reducing the problem to natural language processing on the shorthand of multiple programmers (consider Table 2.1.2). Log anonymization is also troublesome, because sensitive information like usernames is not relegated to distinct fields [16].

**Corruption.** Even on supercomputers with highly engineered RAS systems, like BG/L and Red Storm, log entries can be corrupted. We saw messages truncated, partially overwritten, and incorrectly timestamped. For example, we found many corrupted variants of the following message on Thunderbird (only the message bodies are shown):

```
kernel:  VIPKL(1): [create_mr] MM_bld_hh_mr failed (-253:VAPI_EAGAIN)
```

Some corrupted versions of that line include:

```
kernel:  VIPKL(1): [create_mr] MM_bld_hh_mr failed (-253:VAPI_EAure = no
kernel:  VIPKL(1): [create_mr] MM_bld_hh_mr failed (-253:VAPI_EAGAI
kernel:  VIPKL(1): [create_mr] MM_bld_hh_mr failed (-253:VAPI_EAGsys/mosal_iobuf.c
    [126]: dump iobuf at 0000010188ee7880 :
```

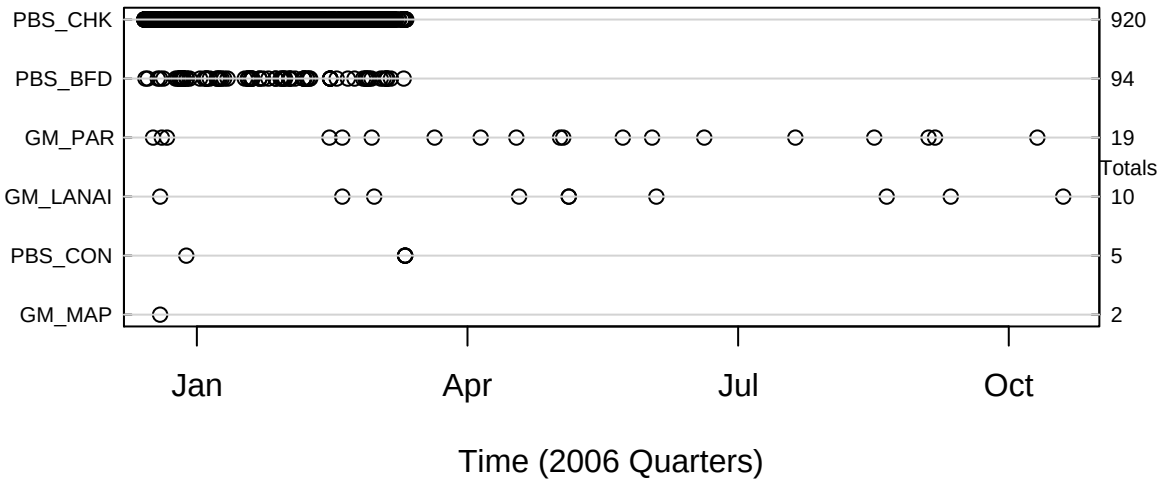



Figure 2.5: Categorized filtered alerts on Liberty over time. The horizontal clusters of PBS_CHK and PBS_BFD messages are not evidence of poor filtering; they are actually instances of individual failures. Specifically, they are the manifestation of the PBS bug described in Section 2.1.3. These two tags are a particularly outstanding example of correlated alerts relegated to different categories.

### 2.1.3 Filtering

A single failure may generate alerts across many nodes or many alerts on a single node. Filtering is used to reduce a related set of alerts to a single initial alert per failure; that is, to make the ratio of alerts to failures nearly one. This section motivates the need for effective filtering and then describes our algorithm, which is based on previous work [27, 28] with some incremental optimizations. Briefly, the filtering removes an alert if any source had generated that category of alert within the last $T$ seconds, for a given threshold $T$. Two alerts are in the same category if they were both tagged by the same expert rule.

#### Motivation for Filtering

During the first quarter of 2006, Liberty saw 2231 job-fatal alerts that were caused by a troublesome software bug in the Portable Batch System (PBS). The alerts, which read `pbs_mom: task_check, cannot tm_reply`, indicated that the MPI rank 0 mom died. Jobs afflicted by this bug could not complete and were eventually killed, but

not before generating the `task_check` message up to 74 times. We estimate that this bug killed as many as 1336 jobs before it was tracked down and fixed (see Figure 2.5).

Between November 10, 2005 and July 10, 2006, Thunderbird experienced 3,229,194 so-called "Local Catastrophic Errors" related to VAPI (the exact nature of many of these alerts is not well-understood by our experts). A single node was responsible for 643,925 of them, of which filtering removes all but 246.

The Spirit logs were largest, despite the system being the second smallest. This was due almost entirely to disk-related alert messages which were repeated millions of times. For example, over a six-day period between February 28 and March 5, there was a disk problem that triggered a total of 56,793,797 alerts. These were heavily concentrated among a handful of problematic nodes. Over the complete observation period, node id sn373 logged 89,632,571 such messages, which was more than half of all Spirit alerts.

### Filtering Algorithm

A temporal filter coalesces alerts within $T$ seconds of each other on a given source into a single alert. For example, if a node reports a particular alert every $T$ seconds for a week, the temporal filter keeps only the first. Similarly, a spatial filter removes an alert if some other source had previously reported that alert within $T$ seconds. For example, if $k$ nodes report the same alert in a round-robin fashion, each message within $T$ seconds of the last, then only the first is kept. Previous work applied these filters serially [27, 28].

Our filtering algorithm, however, performs both temporal and spatial filtering simultaneously; an alert message generated by source $s$ is considered redundant (and removed) if *any source*, including $s$, had reported that alert category within $T$ seconds. This change reduces computational costs (16% faster on the Spirit logs) and is conceptually simpler. We applied this filter to the logs from the five supercomputers using $T = 5$ seconds in correspondence with previous work [9, 27, 28]. The algorithm in pseudocode is given below, where $A$ is the sequence of $N$ unfiltered alerts. Alert $a_i$ happens at time $t_i$ and has category $c_i$. The sequence is sorted by increasing time. The table $X$ is used to store the last time at which a particular category of alert was

reported.

**Algorithm 2.1.1:** LogFilter($A$)

$l \leftarrow 0$
**for** $i \leftarrow 1$ **to** $N$
**do**
- **if** $t_i - l > T$
  - **then** clear($X$)
- $l \leftarrow t_i$
- **if** $c_i \in X$ **and** $t_i - X[c_i] < T$
  - **then** $X[c_i] \leftarrow t_i$
  - **else**
    - $X[c_i] \leftarrow t_i$
    - **output** $(a_i)$

This filter may remove independent alerts of the same category that, by coincidence, happen near the same time on different nodes. For example, node sn373 on Spirit experienced disk problems and output tens of millions of alerts over the course of several days. Coincidentally, another node (sn325) had an independent disk failure during this time. Our filter removed the symptomatic alert, erroneously.

In some cases, serial filtering fails to remove alerts that share a root cause, and which a human would consider to be redundant. The problem arises when the temporal filter removes messages that the spatial filter would have used as cues that the failure had already been reported by another source. Alerts removed by our filter that would be left by serial filters tend to indicate failures in shared resources that were previously noticed by another node. The most common such errors in Liberty, Spirit, and Thunderbird were related to the PBS system.

At most one true positive was removed on any single machine, whereas sometimes dozens of false positives were removed by using our filter instead of the serial algorithm. Limiting false positives to an operationally-acceptable rate tends to be the critical factor in fault and intrusion detection systems, so we consider this trade-off to be justified.

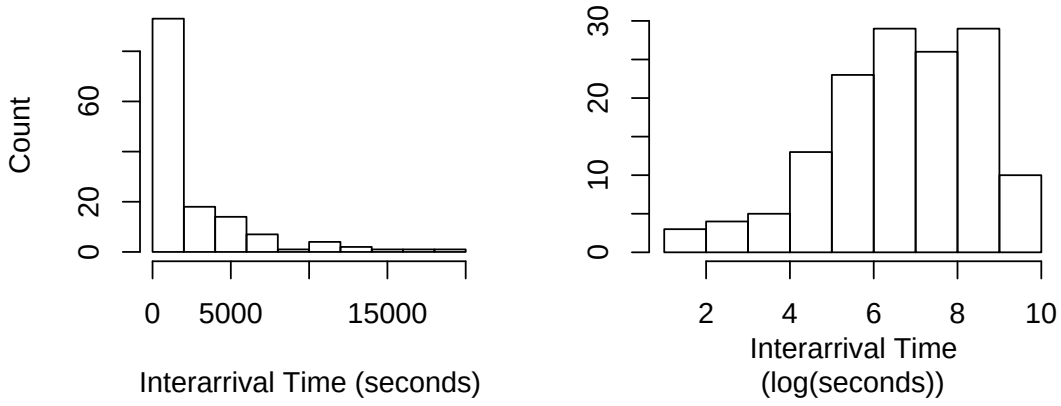
## 2.2 Analysis

Modeling the timing of failure events is a common endeavor in systems research; these models are then used to study the effects of failures on other aspects of the system, such as job scheduling or checkpointing performance. Frequently, for mathematical convenience and reference to basic physical phenomena, failures are modeled as occurring independently (exponentially distributed interarrival times). For low-level failures triggered by such physical phenomena, these models are appropriate; we found that ECC failures (memory errors that were critical, rather than single bit errors) behaved as expected. Figure 2.6 shows these filtered alert distributions on Thunderbird, where the distribution appears exponential and is roughly log normal with a heavy left tail.

For most other kinds of failures, however, this independence is not an appropriate assumption. Failure prediction based on time *interdependence* of events has been the subject of much research [28, 30, 37, 54], and it has been shown that such prediction can be a potent resource for improving job scheduling [43], quality of service [42], and checkpointing [40, 41].

We expected CPU clocking alerts, for instance, to be similar to ECC alerts: driven by a basic physical process. We were surprised to observe clear spatial correlations, and discovered that a bug in the Linux SMP kernel sped up the system clock under heavy network load. Thus, whenever a set of nodes was running a communication-intensive job, they would collectively be more prone to encountering this bug. We investigated this message only after noticing that its occurrence was spatially correlated across nodes.

Through our attempts to model failure distributions, we are convinced that supercomputer failure types are diverse in their properties. Some clearly appear to be lognormal (Figure 2.6(a)), most clearly do not (Figures 2.7(a) and 2.6(b)). In even the best visual fit cases, heavy tails result in very poor statistical goodness-of-fit metrics. While the temptation to select and publish best-fit models and parameters is strong, the most important observation we can make is that such modeling of this data is misguided. The mechanisms and interdependencies of failures must be better



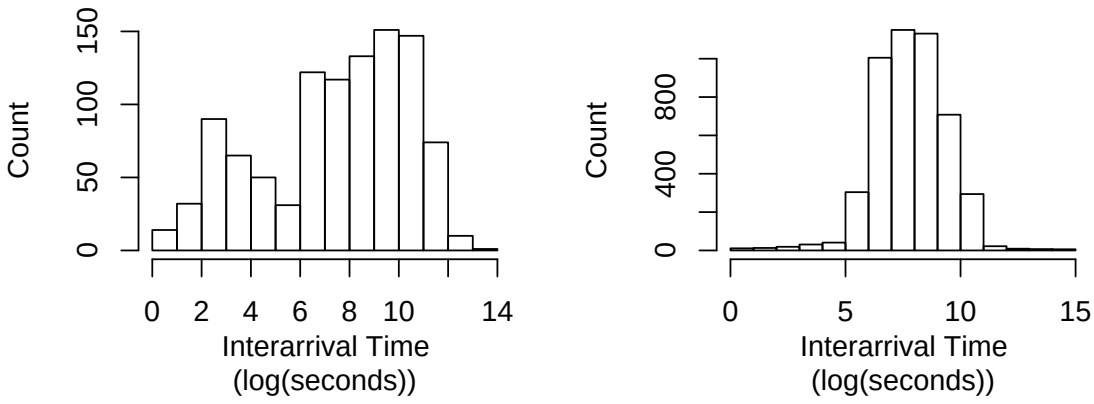

(a) (b)

Figure 2.6: Critical ECC memory alerts on Thunderbird. These data are filtered, but that had little effect on the distribution. Both (a) and (b) are the same data, viewed in different ways. We conclude that these low-level failures are basically independent.

understood before statistical models of their distributions will be of significant use. The merit of a model is dependent on the context in which it is applied; one size does not fit all.

Moreover, whereas the failures in this study have widely varying signatures, previous prediction approaches focused on single features for detecting all failure types (e.g. severity levels or message bursts). Future research should consider ensembles of predictors based on multiple features, with failure categories being predicted according to their respective behavior.

Current filtering algorithms, including ours, suffer from two significant weaknesses. First, they require a mechanism for determining whether two alerts from different sources at different times are "the same" in some meaningful way. We are not aware of any method that is able to confidently state whether two messages that are labeled as different are actually driven by the same underlying process. The second major weakness is that a filtering threshold must be selected in advance and is then applied across all kinds of alerts. In reality, each alert category may require a different threshold, which may change over time. The bimodal distribution visible in Figure 2.7(a) is believed to be a consequence of these shortcomings. One of the modes (the left




(a) (b)

Figure 2.7: The log distribution of interarrival times after filtering suggests correlated alerts on BG/L (a) and largely independent categories on Spirit (b). This illustrates two weaknesses in current filtering algorithms: (1) message tags must represent independent sets of alerts to avoid timing correlations and (2) a single filtering threshold is not appropriate for all kinds of messages.

peak) is attributed to unfiltered redundancy. Figure 2.4 shows an example of inter-tag correlation. On Spirit, the problems enumerated above were not as prevalent after filtering, and the result was the unimodal distribution in Figure 2.7(b).

## 2.3 Lessons Learned

In order to accurately detect, attribute, quantify, and predict failures in supercomputers, we must understand the behavior of systems, including the logs they produce. This chapter presents the results of the broadest system log study to date (nearly one billion messages from five production supercomputers). We consider logs from the BG/L, Thunderbird, Red Storm, Spirit, and Liberty supercomputers (Section 2.1), and we identify 178,081,459 alert messages in 77 categories (Table 2.1.2). In conclusion, we describe how people want to use supercomputer logs, what obstacles they face, and our lessons for overcoming those challenges.

**Detect Faults** We want to identify failures quickly. Most failures are evidenced in logs by a signature (the presence or absence of certain messages), while others leave no sign. We believe such silent failures are rare. Accurate detection and disambiguation requires external information like operational context (Figure 2.1). We suggest logging transitions among operational states (Section 2.1.2). Chapter 3 deals with the challenge of detecting alerts even in the absence of such information.

**Attribute Root Causes** We want to respond to failures effectively, which requires knowing what failed and why. Logging mechanisms themselves may fail, resulting in corrupted or missing messages. Redundant and asymmetric alert reporting necessitates filtering (Section 2.1.3); we advise that future work investigate filters that are aware of correlations among messages and characteristics of different failure classes, rather than a catch-all threshold (Section 2.2). In Chapter 4, we use the correlated generation of surprising messages as a way to infer interactions.

**Quantify RAS** We want to model and improve RAS metrics. Despite the temptation to calculate values like MTTF from the system logs, doing so can be inaccurate and misleading. The content of the logs is a strong function of the specific system and logging configuration; using logs to compare machines is absurd. Even on a single system, the logs change over time, making them an unreliable measure of progress. We recommend calculating RAS metrics based on quantities of direct interest, such as the amount of useful work lost due to failures.

**Predict Failures** We want to predict failures in order to minimize their impact. The mapping from failures to message signatures is many-to-many. Prediction efforts must account for significant shifts in system behavior (Section 2.1.2). Just as filtering would benefit from catering to specific classes of failures, predictors should specialize in sets of failures with similar predictive behaviors (Section 2.2). Our use of influence for cascade detection in Chapter 6 predicts events with similar preceding behaviors.

System logs are a rich, ubiquitous resource worth exploiting. They present many analysis challenges, however, and should not be taken lightly. The lessons in this chapter will lead us closer to our goal: reliable computing for production users.

# Chapter 3

# Alert Detection

In the previous chapter, we examined logs from production supercomputers and found them to be missing certain important information (incomplete) and rife with irrelevant information (noisy). We collaborated with system administrators to perform a laborious tagging of alert messages (the "signal" in the logs). In this chapter, we develop a method for identifying these messages automatically that leverages a crucial insight: similar components running similar workloads tend to generate similar logs. This turns alert detection into an anomaly detection problem. Our resulting method outperforms the system administrators, identifying many alert messages of which they had been previously unaware.

Specifically, we present *Nodeinfo*, an unsupervised algorithm for anomaly detection in system logs. We demonstrate Nodeinfo's effectiveness on data from four of the supercomputer logs from the previous chapter. (We exclude Red Storm because our performance metrics require us to label the logs using data that was not available for that system.) This is the first work to investigate anomaly detection on (several) publicly-available supercomputer system logs, thereby providing a reproducible performance baseline.

The manifestation of a fault in the log is an *alert*; every line in a log has an associated *alert* category, which may be *null*. Lines with a non-null alert category (henceforth *alerts*) are messages that merit the attention of a system administrator, either because immediate action must be taken or because there is an indication of

an underlying problem. Many alerts may be symptomatic of the same *failure*. Failures may be anything from a major filesystem malfunction to a transient connection loss that kills a job (see Chapter 2 for examples). The task of *alert detection* is to automatically separate both new and known alerts from innocuous messages (noise).

In this chapter, we formalize the alert detection task and propose a metric called *binary scoring* that we found to appropriately quantify operational value (Section 3.1). We then present *Nodeinfo*, an unsupervised alert detection algorithm that considers the information entropy of message terms (Section 3.2). Nodeinfo may be applied to any timestamped and tokenizable text log; it requires no system-specific information. We demonstrate that Nodeinfo can effectively identify alerts using statistical properties of the log (Sections 3.3 and 3.4).

The analysis gives us deeper insight into the logs that even months of prior study did not provide. First, we identify and confirm ten new alert categories, yielding new alert messages on all four systems (Section 3.3.1). Second, we observe that similar computers, correctly executing similar workloads, tend to generate similar logs (Section 3.3.3), and demonstrate how statistical algorithms can exploit this fact to obtain substantial performance improvements (Section 3.4). Specifically, our algorithm ultimately achieves up to seven times baseline performance on some workloads.

Nodeinfo is currently in production use on at least three supercomputers. The data sets and code are public [64, 65], so the results in this chapter are verifiable and reproducible; this work provides a performance baseline for alert detection. These first steps have already improved the system administration task for the machines under study, and we are confident that further efforts will be similarly rewarded.

## 3.1 The Challenge

Let a *log* be a sequence $L$ of lines $l_1$ through $l_N$, where $N$ is the total number of *messages*: $L = (l_1, l_2, \ldots, l_N)$. Each line $l_i$ consists of a sequence of characters broken up into tokens $s_1$ through $s_M$ according to some delimiting sequence of characters (we use whitespace): $l_i = (s_1, s_2, \ldots, s_M)$. Let $s_{i,m}$ denote the $m^{th}$ token of line $i$. $M$ may be different from line to line, and $s_m$ need not have the same semantic interpretation

throughout the log. We calculate and prepend certain important tokens. Let $t$ be the `utime` (universal time in seconds) for the line, and so let $t_i$ be the time of line $i$. In addition, let $c_i$ be the computer (hereafter *node*) that generated the line.

We decompose logs into *nodehours*: all lines from a single node $c$ in one-hour intervals corresponding to wall-clock time. Such lines are said to be *contained* by (or *in*) their corresponding nodehour; the *count* of a nodehour is the number of messages it contains. We define $H_j^c$, the $j^{th}$ nodehour for node $c$, to be $H_j^c = \{l_i \mid c_i = c \wedge j \leq t_i/3600 < j + 1\}$.

Decomposing logs into documents by source and time reflects how many system administrators divide and conquer logs, as learned through the trial-and-error of experience. Individual lines lack the context to sufficiently characterize a message, whereas huge log dumps with interleaved node output can be difficult to mentally parse. A nodehour provides a happy medium between these extremes, and we found it to be a useful abstraction for detecting alerts. Implicitly, nodehours are chunks of data that we expect to be relatively uniform in content over time—anomalous nodehours are noteworthy events.

### 3.1.1 Objective

Ultimately, the goal of reliability research is to minimize unscheduled downtime. An alert detector can facilitate this goal by accurately identifying when and where alerts are generated, so that remedial or preventive action may be taken. Every log line is categorically either an alert or not, so nodehours, which consist of lines, can also be categorized by whether they contain alerts or not. A natural objective is to automatically rank nodehours by the probability that they contain alerts.

In the original data sets, every line is *tagged* with an alert category using a combination of expert rules and manual labeling. In this chapter, we extend that tagging, due in part to the results from our alert detection methods (Section 3.3.1). The message tags are used exclusively for quantifying the effectiveness of our alert detection methods: the methods themselves ignore the tags entirely.

### 3.1.2 Metrics

Call $H_j^c$ an *alert nodehour* if it contains at least one alert. An alert detection algorithm outputs a list of nodehours, sorted in decreasing order of the probability that each is an alert nodehour. Let $R_k$ be the union of nodehours formed by taking the top $k$ nodehours from this output list.

Scoring a ranking of nodehours depends on the definition of what constitutes a true positive (TP), false positive (FP), true negative (TN), and false negative (FN). We experimented with several such definitions before determining that *binary scoring*, described below, is most useful in practice. This conclusion is related to the fact that some faults are *bursty*, meaning they produce numerous alert messages in a short period of time. Although the majority of fault types are not bursty, our alternative metrics disproportionately reward discovery of bursty alerts. (Many naïve algorithms seemed excellent, often near-optimal.)

The *binary scoring* metric treats nodehours as atomic, considering only whether or not each nodehour $H_j^c$ is an alert nodehour. Nodehours are categorized as follows:
$\text{TP} = \{H_j^c \in R_k \mid \exists l_i \in H_j^c \text{ s.t. } l_i \text{ is an alert}\}$
$\text{FP} = \{H_j^c \in R_k \mid \forall l_i \in H_j^c,\ l_i \text{ is not an alert}\}$
$\text{TN} = \{H_j^c \notin R_k \mid \forall l_i \in H_j^c,\ l_i \text{ is not an alert}\}$
$\text{FN} = \{H_j^c \notin R_k \mid \exists l_i \in H_j^c \text{ s.t. } l_i \text{ is an alert}\}$

A set of nodehours, $R_k$, yields a single value each for precision ($\frac{TP}{TP+FP}$), recall ($\frac{TP}{TP+FN}$), and the standard F1 measure ($\frac{2*TP}{2*TP+FP+FN}$). Binary scoring spurred us to investigate false positives, thereby finding new alert types that other metrics had obscured; some of these alerts were previously unknown even to the system administrators.

Solutions are driven by the metrics used to assess them. For alert detection, the scoring method must be chosen carefully to prevent bursty alerts from eclipsing the more elusive ones. In addition to accurately reflecting the true value of an algorithm, binary scoring accomplishes what per-alert category threshold filtering [27, 54] does not: captures and filters cross-category temporal correlations. Based on our experience in this study, and following the lead of previous work [66], we recommend using binary scoring on nodehours.

### 3.1.3 Optimal and Baseline

The theoretical, optimal algorithm (OPT) outputs exactly the list of alert nodehours and appears implicitly in all precision-recall plots as a horizontal line at a precision of one. In addition to OPT, we compute scores for a baseline that represents the predominant practice of system administrators. This Bytes baseline simply ranks nodehours by the number of bytes of message data they contain, from largest to smallest. This practice is based on the fact that some alert categories are bursty, and thus the highest-byte nodehours often do contain alerts.

## 3.2 Nodeinfo

The motivating premise of Nodeinfo is that similar computers correctly executing similar workloads should produce similar logs, in terms of content (i.e., line tokens). Nodeinfo is universally applicable, in the sense that it can be computed on any tokenizable log with timestamps. Nodeinfo does not train on labeled data; it is unsupervised. The results represent a performance baseline for alert detection without incorporating system-specific information. The development of Nodeinfo, as well as its performance on a small subset of the data considered in this chapter, is detailed elsewhere [66]. This chapter contributes examination of its effectiveness on multiple systems, larger systems, and over longer time ranges (together resulting in nearly two orders of magnitude more data). In addition, it examines the effects of using different sized sliding windows, as done in practice.

The first step is to compute how much information each token conveys regarding the computer that produced it. Let $W$ be the set of unique *terms* formed by concatenating each line token with its position $m$ in the line ($w_m = m, s_{i,m}$), and let $C$ be the total number of nodes. Let $\mathbf{X}$ be a $|W| \times C$ matrix such that $x_{w,c}$ is the number of times term $w$ appears in messages generated by node $c$. Towards understanding how unevenly each term is distributed among nodes, let $G$ be a vector of $|W|$ weights where $g_w$ is equal to 1 plus term $w$'s Shannon information entropy [6]. Specifically, $g_w = 1 + \frac{1}{\log_2(C)} \sum_{c=1}^{C} p_{w,c} \log_2(p_{w,c})$, where $p_{w,c}$ is the number of times term $w$ occurs

on node $c$ divided by the number of times it occurs on *any* node ($p_{w,c} = \frac{x_{w,c}}{\sum_{c=1}^{C} x_{w,c}}$). Thus, a term appearing on only a single node receives a weight of 1, and a term appearing the same number of times on all nodes receives a weight of 0.

The second step ranks nodehours according to how many high-information terms each contains. Let $H$ be the set of all nodehours and let $\mathbf{Y}$ be the $|W| \times |H|$ matrix where $y_{w,c,j}$ is the number of times term $w$ occurs in nodehour $H_j^c$. The Nodeinfo value for each nodehour is then calculated as

$$\text{Nodeinfo}(H_j^c) = \sqrt{\sum_{w=1}^{|W|} (g_w \log_2(y_{w,c,j} + 1))^2}.$$

Nodehours are then ranked by decreasing Nodeinfo value. Those containing high-information terms will be ranked high, and those containing low-information terms (even a great number of them) will be ranked low. These calculations are modeled after the "log.entropy" weighting scheme [6], where term entropy is calculated over node documents and then applied to all nodehour documents in the corpus.

We now describe two practical considerations important for reproducibility of our results. First, we exclude all $m = 1$ terms (timestamps) in order to decrease the false positive rate [66]. Second, we institute a minimum support threshold of 2. Most terms are infrequent, so this significantly reduces memory overhead and has little impact on the outcome. For instance, terms with a support of 1 must each have a weight of 1, but can contribute no more than the square root of their total number to a nodehour's Nodeinfo($H_j^c$) magnitude. One class of terms eliminated in this manner is hexadecimal addresses, which rarely facilitate alert detection. We do not evaluate the actual impact of this threshold on the results due to computational limitations.

## 3.3 Results

We ran Nodeinfo offline on data from four of the supercomputers from Chapter 2: Liberty, Spirit, Thunderbird, and Blue Gene/L (BG/L). In these initial tests, the algorithm did not significantly outperform Bytes; moreover, it was far from optimal.

| Alert Type/Cat. | Count | Nodehours | Example Message Body (Anonymized) |
|---|---|---|---|
| | | | **Affected Systems: Thunderbird, Spirit, and Liberty** |
| H / CHK_COND | 3,948,364 | 66 | kernel: [hex] has CHECK CONDITION, sense key = 0x3 |
| S / EXT_INODE | 1,299,603 | 47 | kernel: EXT3-fs error [...] unable to read inode block - [...] |
| H / HDA_NR | 883,399 | 1846 | kernel: hda: drive not ready for command |
| H / HDA_STAT | 883,398 | 1846 | kernel: hda: status error: status=[...] |
| S / PBS_U09 | 437,776 | 199 | pbs_mom: Unknown error 15009 (15009) in job_start_error from node [IP:port], [job] |
| S / PBS_EPI | 53,647 | 1192 | pbs_mom: scan_for_exiting, system epilog failed |
| S / CALL_TR | 40,810 | 839 | kernel: Call Trace: [<[...]>] net_rx_action [kernel] [...] |
| S / PBS_U23 | 5177 | 8 | pbs_mom: Unknown error 15023 (15023) in job_start_error from node [IP:port], [job] |
| | | | **Affected System: Blue Gene/L** |
| H / DDR_STR | 243 | 241 | ddr: Unable to steer [...] consider replacing the card |
| H / DDR_EXC | 41 | 41 | ddr: excessive soft failures, consider replacing the card |

Table 3.1: Additional actionable alert messages discovered via our algorithms.

We now describe the insights that allow us, in Section 3.4, to improve performance several-fold.

### 3.3.1 Data Refinement

We investigated the initially mediocre performance by scanning the output nodehour lists for false positives, starting with the most highly-ranked nodehours that ostensibly contained no alerts. In these nodehours, we discovered several new alert types that had been incorrectly assigned null alert tags. Using the same rigorous verification process as was employed to tag the original alerts, we updated the data sets with these new alert types. This process involves discussions with the system administrators and a characterization of the alerts that allows us to identify them elsewhere in the log. We also discovered 80 lines erroneously tagged as alerts in the original data. Two were test scripts run on Spirit by an administrator, and the other 78 (on Thunderbird) appear

to have been the result of a buggy tagging script. The new alerts are summarized in Table 3.1, similar to Table 2.1.2 in Chapter 2. "Cat." is the alert category. Types are H (Hardware) and S (Software). Bracketed text indicates omitted information; a bracketed ellipsis indicates sundry text. In all, we discovered ten alert categories, containing 7,552,458 new alert messages across 6325 nodehours.

One might speculate whether such alerts could have been discovered via inspection, such as by selecting and reading random nodehours. Years of intense scutiny by the system administrators, and later by us, failed to elucidate the alerts discovered via our automated method. Thus, we believe such speculation is idle; our information-theoretic algorithm revealed new alert categories with great efficiency, and the administrators have since incorporated these alerts into their production detection infrastructure.

### 3.3.2 Tagging Limitations

In addition to the ten alert categories enumerated in Table 3.1, our analysis revealed dozens of other alert categories that were more challenging to incorporate into our current tagging framework. Whether or not certain messages are alerts may depend on (i) the rate at which the messages were generated (rate-sensitive), (ii) proximate messages or the operational status of the node (context-sensitive), or (iii) whether the corresponding remedy is actually known or elected (non-actionable).

Per-message alert tagging is straightforward (linewise regular expressions) and precise (exact time and source of an alert). Furthermore, the use of linewise tagging in this chapter is consistent with prior work [28, 33]. Nevertheless, limitations of our tagging tools and a poor understanding of rate thresholds obliged us to exclude rate- and context-sensitive messages as alerts. There are reasons to expect that including them would improve the performance of our techniques. System administrators have advised us that non-actionable alerts still merit their attention, and so we treat them on par with actionable alerts for scoring.

| System | Counts | C % | A/IO % | O % |
|---|---|---|---|---|
| **Blue Gene/L** | Total: 1,816,627 | 87.08 | 12.10 | 0.8200 |
| (8.48% alerts) | Alert: 154,014 | 45.42 | 54.52 | 0.0600 |
| **Thunderbird** | Total: 15,255,833 | 89.50 | 0.6603 | 9.840 |
| (0.163% alerts) | Alert: 24,877 | 85.81 | 0.0764 | 14.11 |
| **Spirit** | Total: 6,731,957 | 98.76 | 0.3894 | 0.8506 |
| (0.207% alerts) | Alert: 13,933 | 93.12 | 5.828 | 1.052 |
| **Liberty** | Total: 1,820,433 | 96.07 | 1.492 | 2.438 |
| (0.282% alerts) | Alert: 5139 | 97.90 | 0.1946 | 1.905 |

Table 3.2: Distribution of total and alert nodehours across node types. 'C' is Compute, 'A/IO' is Admin and IO, and 'O' is Other.

### 3.3.3 Similar Nodes

Statistical anomaly detection algorithms like Nodeinfo compare a sample against a reference distribution and measure the variation from "normal"; such algorithms perform better when the reference distribution is from the same homogenous population as the sample. One way to define homogeneous groups is by function. In the context of large clusters and supercomputers, different nodes serve different functions: computation, administration, communication, etc.

A natural question, therefore, is whether Nodeinfo would perform better when run on functionally homogeneous groups of nodes, rather than all nodes, together. To test this, we ran Nodeinfo on logs from all nodes, and on functionally similar subsets independently, but scored all results using the full number of alerts in all the logs. When considering only compute nodes on Liberty, the Nodeinfo algorithm achieves a maximal F1 score that is seven times better than when non-compute nodes are included. (F1 is a popular summary metric, ranging from 0 (bad) to 1 (good) that weights false positives and false negatives equally.) In other words, even when we handicap Nodeinfo by showing it only logs from the compute nodes, its score against the entire data set improves. Although we omit these offline results, the online experiments show similar improvements (Section 3.4).

Table 3.2 shows the distribution of nodehours from each functional group: compute nodes, administrative nodes (Thunderbird, Spirit, and Liberty), IO nodes (BG/L), and other nodes. Columns 4–6 give the percent contribution of that functional group

to the total number of nodehours (first row) and the number of alert nodehours (second row). For example, only 12.1% of the nodehours on BG/L were from I/O nodes, but this functional group contributed 54.52% of the alert nodehours. Considering that only 3.9% of Liberty's nodehours were from non-compute nodes, the significant impact of their exclusion on Nodeinfo is noteworthy. In other words, when we group the nodehours by functional group, we create homogeneous backgrounds against which Nodeinfo is better able to identify the anomalies. These data support our claim that similar computers (compute nodes) tend to generate similar logs.

## 3.4 Online Detection

The offline techniques are valuable for exploring the data, but a production setting requires online detection with low latency. In this section, we modify Nodeinfo to operate using a sliding, bounded history window. Furthermore, we run the algorithm on major functional groups individually, to evaluate the impact of leveraging our observation regarding similar nodes.

We use a "sticky" sliding window to compute the Nodeinfo score for the current nodehours: for reasons of efficiency, this window is not of fixed size; it always starts at midnight $W-1$ days prior, for a window size of $W$ days. For example, if $W = 30$, then all nodehours on January 30th will use data generated since January 1st at 12:00 AM. Thus, $W$ is an upper bound on the amount of history considered in the computation. We consider windows of 30, 60, and 90 days. For consistency, the first 90 days of data are omitted from scoring all online experiments.

To evaluate our similar nodes hypothesis, we divide the logs into functional groups (see Table 3.2) and run Nodeinfo on each group; the resulting lists of ranked nodehours are then scored against *all* functional groups. (Alerts in other groups are automatically false negatives.) Results from BG/L are plotted in Figure 3.1. Detection performance on the IO nodes in isolation exceeds that of detection over the log as a whole, even when dropping alerts from every other functional group on the floor. The compute nodes group on Spirit (Figure 3.2) yields area under the curve more than twice that of both Bytes and Nodeinfo run on the entire log, even without considering

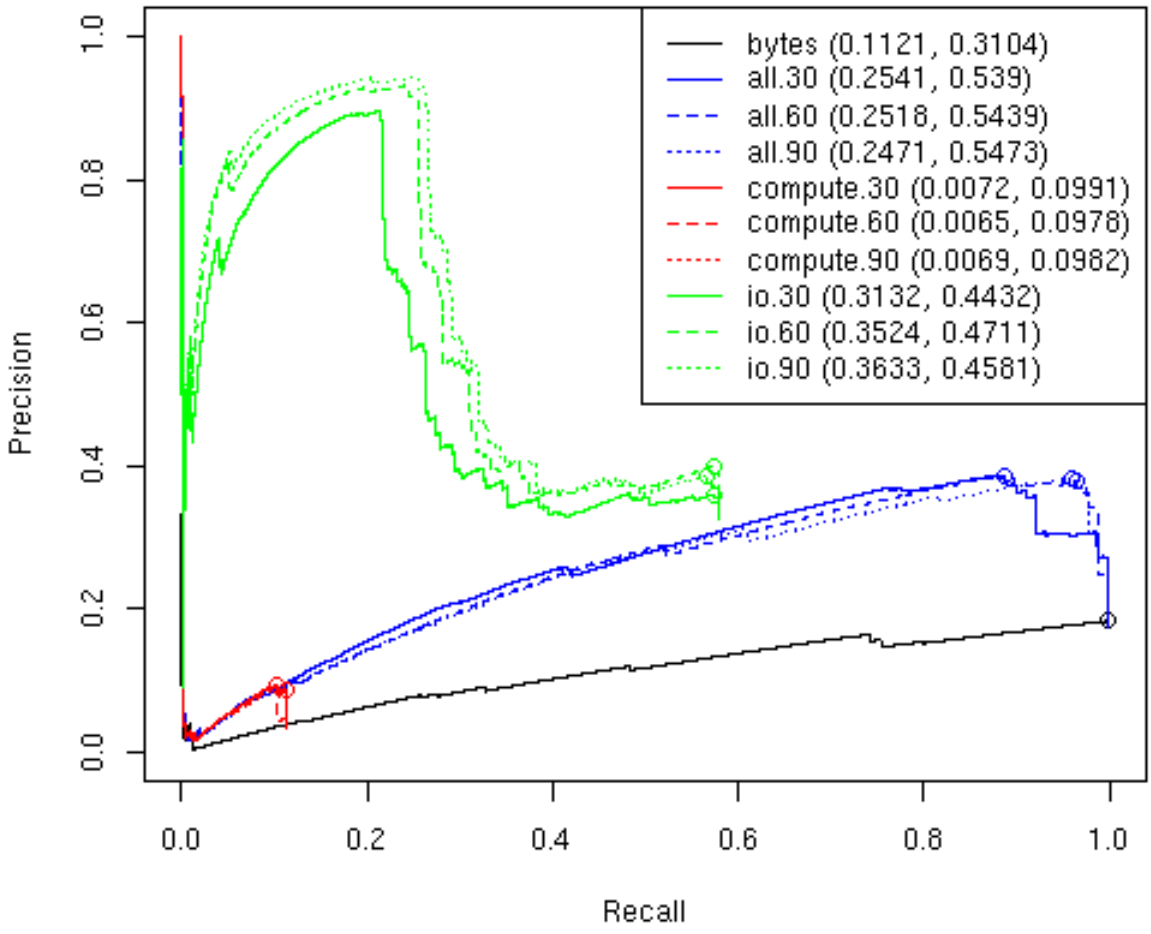



Figure 3.1: Precision-recall curves for the online Nodeinfo detector on Blue Gene/L. The legend indicates the functional group, window size, area under the curve, and maximal F1: `group.window (area, F1)`.

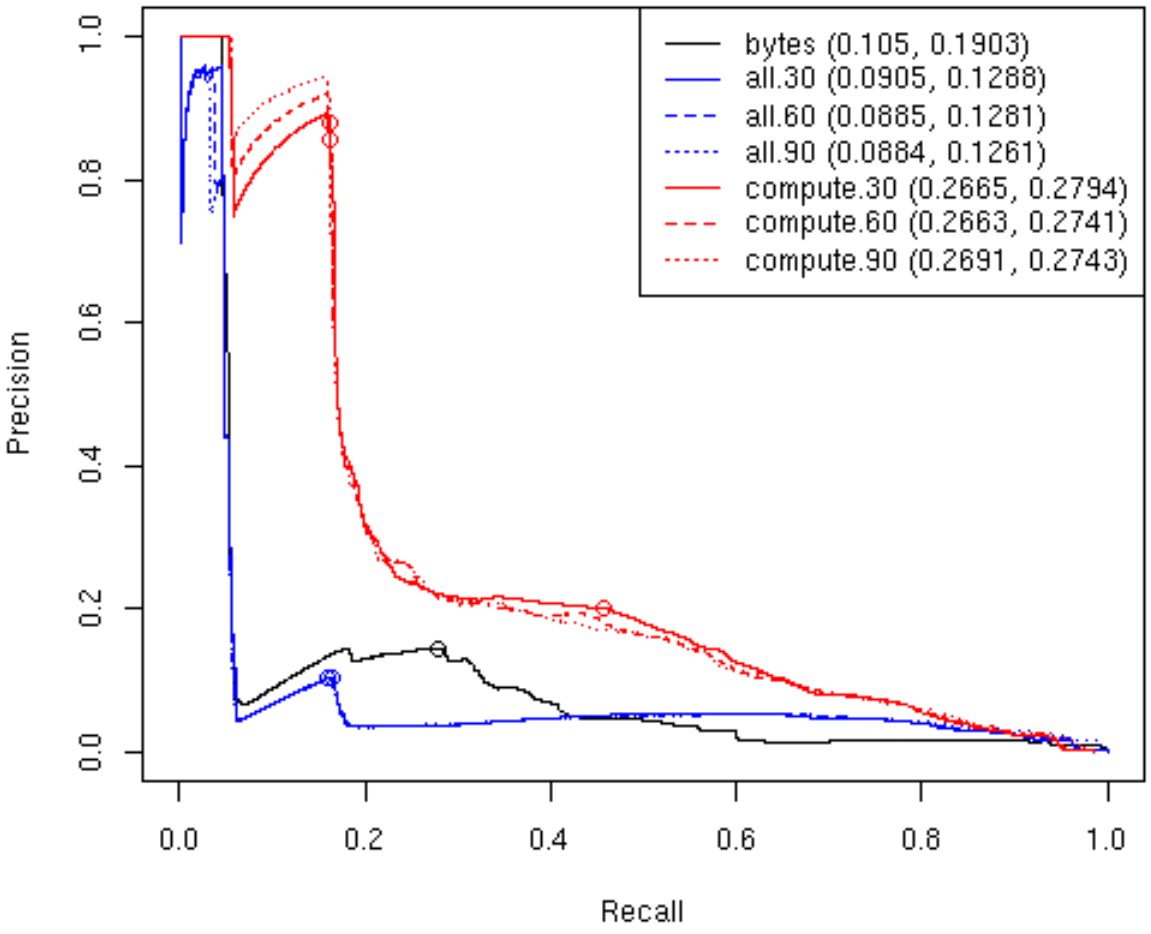



Figure 3.2: Precision-recall curves for the online Nodeinfo detector on Spirit. The legend indicates the functional group, window size, area under the curve, and maximal F1: `group.window (area, F1)`.

the remainder of the data. The results from BG/L and Spirit are representative.

For comparison, consider the area under the curve metric when the results in Figures 3.1 and 3.2 are scored against only their respective functional groups: on the BG/L IO nodes, Nodeinfo achieves 0.63 of OPT; on the compute nodes, the metric improves nearly ten-fold. The critical conclusion is that we can leverage the homogeneity of large systems to dramatically improve the quality of alert detection.

Our results show that (i) Nodeinfo is an improvement over Bytes, (ii) Nodeinfo performs better on homogenous functional groups than on all the log at once, and (iii) larger window sizes yield marginal improvements, suggesting both that the logs are changing slowly enough for the past few months to reflect the future and that the computationally inexpensive parameters yield competitive results.

## 3.5 Contributions

The main goal of reliability research for high performance computing is to minimize unscheduled downtime. To do so, we must reduce the time that system administrators spend investigating new fault types and performing root cause analysis. Better techniques for alert detection are an important step toward more efficient system management, automatic fault prediction and response, and greater overall reliability.

In this chapter, we formalized the alert detection problem; presented Nodeinfo, a general, unsupervised alert detection algorithm; and quantitatively demonstrated its effectiveness on 81 GB of public supercomputer system logs. Our most salient insight into the alert detection problem is that similar nodes running similar workloads tend to generate similar logs—we can better identify anomalous behavior when normal behavior appears more uniform. The most compelling evidence of our success is that we discovered several new alert categories that had eluded experts for years and that our online implementation of Nodeinfo [65] is in production use on Red Storm, Thunderbird, and Liberty. According to the administrators of these systems, "[our method] has automatically detected, and more importantly isolated, a wide range of problems," and they have found it to be "a useful diagnostic tool." Thus, our work has already had a positive operational impact on the systems we studied.

# Chapter 4

# Influence

The previous chapters focused on what information system logs contain, the challenges associated with analyzing them, and a method for quantifying how statistically surprising parts of the log are as a means for detecting misbehavior. However, when something goes wrong in a complex production system—a performance glitch, a strange result, or an outright crash—detection alone is insufficient. It is also crucial to identify the components that are likely sources of the misbehavior.

In this chapter, we assume a method for quantifying surprise—converting the logs generated by components into signals of surprise over time—and turn our attention toward how to use such signals to understand the relationships between those components. Specifically, we define *influences* as a class of component interactions that includes direct communication and resource contention. We show that understanding the existence and strength of interactions, even without knowledge of the underlying mechanism, can be valuable for problem diagnosis.

A fundamental difficulty is that the costs of instrumentation in production systems are often prohibitive. Significant systems are invariably constructed from many interacting subsystems, and we cannot expect to have measurements from every component. In fact, in many systems we will not even know of all the components or of the interactions among the components we do know. Our influence method generates a potentially partial diagnosis from whatever data is available.

Our method requires only that some of the *components* in the system are instrumented to generate timestamped measurements of their behavior. The type of measurements may depend on the type of component (e.g., a laser sensor may be instrumented differently than a hard disk). Thus, we need a way to compare measurements of different components in a uniform way. We address this issue, and the related question of how to summarize different kinds of measurements from a single component, by mapping all components' behavior to a single dimension: surprise. That is, our method quantifies how anomalous individual component behavior is, as an *anomaly signal*, using deviation from a model of normal component behavior. An important feature of our anomaly signals is that they are real-valued, meaning that the degree to which a component's behavior is anomalous is retained, rather than the common approach of discretizing behavior into "normal" and "abnormal".

When two anomaly signals are correlated, meaning that two components tend to exhibit surprising behavior around the same time, we say that the components share an *influence*. This correlation can arise from a number of interactions, including direct communication and contention for a shared resource. Not all interactions are instantaneous, so we use effect delays—how long it tends to take an anomaly in one component to manifest itself in another—to establish directionality. Correlation is a pairwise relationship and delay is directional, so the most natural structure to summarize influence is a graph. A Structure-of-Influence Graph (SIG) encodes strong influence as an edge between components, with directionality to represent a delay.

Passively collected data, if devoid of hints like "component A sent a message to component B," cannot be used to infer causality: the strongest possible mathematical statement is that the behavior of one component is correlated with another. An advantage of using statistical correlation is that it enables asking "what-if" queries, after the fact. For example, it is easy to add a new "component" whose anomaly signal is large around the time bad behavior was observed. Other, real, components that share influence with the synthetic component are likely candidates for contributors to the problem.

Our goal is to generate a structure, informed by models of component behavior, that enables a user to more easily answer prediction and diagnosis questions. The

influence method has several desirable properties:

- Building a SIG requires no intrusive instrumentation; no expert knowledge of the components; and no knowledge about communication channels (e.g., the destination of a message), shared resources, or message content. Our method is passive and can treat components as black boxes.
- Influence describes correlation, not causality. A key feature of our approach is to drop the assumption that we can observe all component interactions and focus on the correlations among behaviors we can observe.
- By working directly with a real-valued, rather than binary, anomaly signal, our method degrades gracefully when data is noisy or incomplete.
- Our experimental results show that SIGs can detect influence in complex systems that exhibit resource contention, loops and bidirectional influence, time-delayed effects, and asynchronous communication.

In this chapter, we present the influence method and work through an example (Section 4.1); perform several controlled experiments using a simulator (Section 4.2) to explore parameters like message drop rate, timing noise, and number of intermediate components; describe the central case study of the chapter, how we took passively collected measurements from two autonomous vehicles and built SIGs that enabled us to identify the source of a critical bug (Section 4.3); and briefly present a significantly different second example by isolating a bug in a production supercomputer (Section 4.4).

## 4.1 The Method

This section describes how to construct and interpret a Structure-of-Influence Graph (SIG). The construction process consists of four steps: decide what information to use from each component (Section 4.1.1), measure the system's behavior during actual operation as *anomaly signals* (Section 4.1.2), compute the pairwise cross-correlation between all components' anomaly signals to determine the strength and delay of

correlations (Section 4.1.3), and construct a SIG where the nodes are components and edges represent the strength and delay of correlations between components (Section 4.1.4). We later apply these techniques to idealized systems (Section 4.2) and real systems (Sections 4.3 and 4.4).

### 4.1.1 Modeling

Log data tends to contain irrelevant information (noisy) and lack important information (incomplete); a *model* is a means of filtering the noise, amplifying the signal, and encoding external knowledge. For example, a user might know that a particular sequence of messages is surprising and build that into the model. Even in the absence of such expert knowledge, a model can express the intuition that deviation from typical measurements is surprising; we use such models throughout this chapter.

The choice of component models determines the semantics of the anomaly signal and, consequently, of the SIG. For example, if we model a program using the distribution of system call sequences and model a memory chip using ECC errors, then the relationship of these components in the resulting SIG represents how strongly memory corruption influences program behavior, and vice versa. There is not, therefore, a single correct choice of models; for a particular question, however, some models will produce SIGs better suited to providing an answer.

We have found two models particularly useful in practice: one based on message timing, which is useful for systems where timing behavior is important (e.g., embedded systems) and at least some classes of events are thoroughly logged (see Section 4.3), and one based on the information content of message terms, useful for systems where logging is highly selective and ad hoc (see Section 4.4). The timing model keeps track of past interarrival times (the first difference of the timestamps, meaning the difference between each timestamp and the previous one) and computes how "surprising" the most recent spacing of messages is (see Section 4.1.2); the term entropy model looks at the distributions of message contents using the algorithm from Chapter 3.

### 4.1.2 Anomaly Signal

We quantify the behavior of components in terms of surprise: the *anomaly signal* $\Lambda_j(t)$ describes the extent to which the behavior of component $j$ is anomalous at time $t$. The instantaneous value of the signal is called the *anomaly score*. Let $\Lambda(t) = 0$ for any $t$ outside the domain of the anomaly signal. We require that $\Lambda_j(t)$ has finite mean $\mu_j$ and standard deviation $\sigma_j$.

The anomaly signal should usually take values close to the mean of its distribution—this is an obvious consequence of its intended semantics. The distance from the mean corresponds to the extent to which the behavior is anomalous, so values far from the mean are more surprising than those close to the mean.

The user defines what constitutes surprising behavior by selecting an appropriate model. For example, one could use deviation from average log message rate, degrees above a threshold temperature, the divergence of a distribution of factors from an expected distribution, or some other function of measurable, relevant signals.

#### Computing the Anomaly Signal

In this section, we discuss the mechanics of computing the anomaly signal $\Lambda_j(t)$ for the timing model mentioned in Section 4.1.1. We describe the offline version.

Let $S$ be a discrete signal from some component, consisting of a series of time (non-decreasing timestamp) and value pairs: $S = \langle (t_0, v_0), (t_1, v_1), \ldots, (t_s, v_s) \rangle$.

Individually, denote $S(i) = (t_i, v_i)$, $T(i) = t_i$, and $V(i) = v_i$. This work gives special attention to the case when $V(i)$ is the first difference of the time stamps (interarrival times): $V(i) = T(i) - T(i-1)$ and $V(0) = \phi$ (null).

To compute anomaly signals, we compare a histogram of a recent window of behavior to the entire history of behavior for a component. Let $h$ be the (automatically) selected bin width for the histogram (in seconds), let $w$ be the size of the recent history window in number of samples, and let $k = \lceil \frac{\max v_i - \min v_i}{h} \rceil$ be the number of bins. For each bin $H(j)$ in the historical histogram, count the number of observations $V(i)$ such that $jh \leq V(i) < jh + 1$. Let $R(T(i))$ be the analogous histogram computed from the previous $w$ samples, ending with $V(i)$. Note that $R(T(i))$ is not defined

for the samples $V(1)$ through $V(w)$. Let $H'$ and $R'(t)$ be the corresponding probability distributions, where the count in each bin is divided by the total mass of the histogram; $H$ has a mass of $s-1$ and $R(t)$ has a mass of $w$.

Compute the Kullback-Leibler divergence [25] between each recent distribution $R'(t)$ and the historical distribution $H'$, producing the anomaly signal $\Lambda(t)$:

$$\Lambda(t) = D_{KL}(R'(t)||H') = \sum_{k \in R'(t)} R'(t,k) \log_2 \frac{R'(t,k)}{H'(k)}.$$

Intuitively, KL-divergence is a weighted average of how much the fraction of measurements in bin $R'(t,k)$ differs from the expected fraction $H(k)$.

After computing $\Lambda_j(t)$ for each component, we store the sampled signals as an $n \times m$ matrix, where $n$ is the number of components and $m$ is the number of equispaced times at which we sample each anomaly signal. We then process these matrices as described starting in Section 4.1.3. Observe that, having represented the system as a set of anomaly signals, the rest of our method is system-independent.

### 4.1.3 Correlation and Delay

For each pair of components $(i,j)$, compute the normalized cross-correlation of their anomaly signals:

$$(\Lambda_i \star \Lambda_j)(t) \equiv \int_{-\infty}^{\infty} \frac{[\Lambda_i(\tau) - \mu_i][\Lambda_j(t+\tau) - \mu_j]}{\sigma_i \sigma_j} d\tau. \tag{4.1}$$

The function $(\Lambda_i \star \Lambda_j)(t)$ gives the Pearson product-moment correlation coefficient of the anomaly signals of components $i$ and $j$, with an offset, or *delay*, of $t$ time steps; it is the correlation of the two signals if $\Lambda_i$ were delayed relative to $\Lambda_j$ by $t$. Consider two hypothetical components, $A$ and $B$, whose cross-correlation is plotted in Figure 4.1. There is a peak at $t = -100$ because anomalies on $A$ tend to appear 100 units of time before they do on $B$. This plot would be represented in the SIG by an edge $A \to B$.

The resulting $O(n^2)$ cross-correlation functions of an $n$-component system are the primary output of our analysis. (It is worth noting that Project5 uses a form

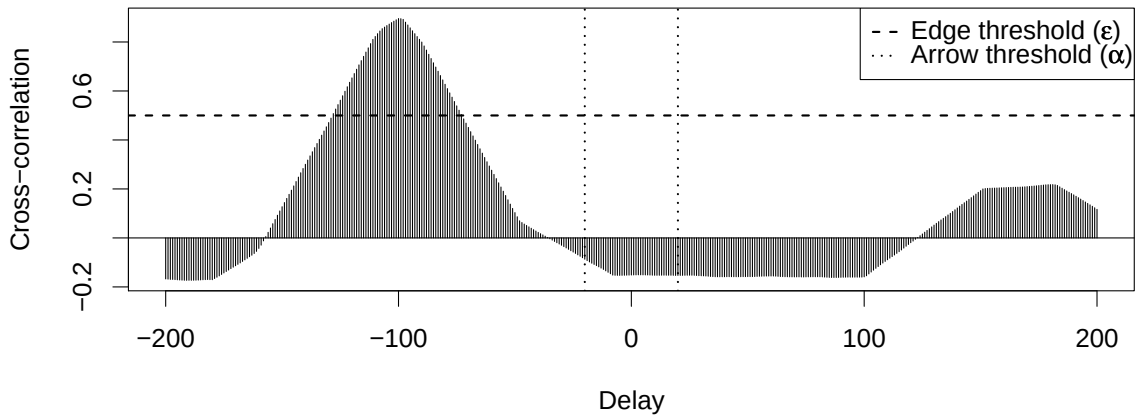



Figure 4.1: The normalized cross-correlation between components $A$ and $B$.

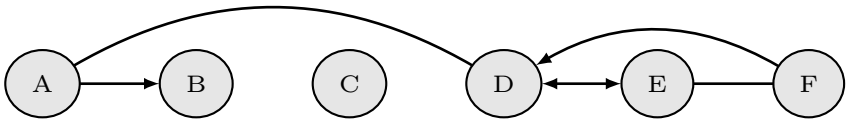



Figure 4.2: The Structure-of-Influence Graph for a system that includes $A$ and $B$.

of signal correlation with communication events to compute dependencies [2].) In general, however, these correlation vectors contain too much information to present a useful view of a system; we distill these data into simpler forms. First, we represent two salient features of the functions (specific extrema values and positions) as two order $n \times n$ matrices: $\mathbf{C}$ and $\mathbf{D}$. Second, in Section 4.1.4, we transform these matrices into a SIG. The SIG is often the shortest path to insights about a system, but the underlying data is always available for inspection or further analysis.

We now construct the correlation matrix $\mathbf{C}$ and delay matrix $\mathbf{D}$ from the cross-correlation functions. Consider a particular pair of components, $i$ and $j$. Let $d_{ij}^-$ and $d_{ij}^+$ be the offsets closest to zero, on either side, at which the cross-correlation function is most extreme:

$$d_{ij}^- = \max(\text{argmax}_{t \leq 0}(|(\Lambda_i \star \Lambda_j)(t)|)) \text{ and}$$
$$d_{ij}^+ = \min(\text{argmax}_{t \geq 0}(|(\Lambda_i \star \Lambda_j)(t)|)),$$

where $\text{argmax}_t f(t)$ is the set of $t$-values at which $f(t)$ is maximal. (One could also select the delays furthest from zero, if that is more appropriate for the system under

study.) Next, let $c_{ij}^-$ and $c_{ij}^+$ be the correlations observed at those extrema: $c_{ij}^- = (\Lambda_i \star \Lambda_j)(d_{ij}^-)$ and $c_{ij}^+ = (\Lambda_i \star \Lambda_j)(d_{ij}^+)$.

Let entry $\mathbf{C}_{ij}$ of the correlation matrix be $c_{ij}^-$ and let $\mathbf{C}_{ji}$ be $c_{ij}^+$. (Notice that $c_{ij}^+ = c_{ji}^-$.) Similarly, let entry $\mathbf{D}_{ij}$ of the delay matrix be $d_{ij}^-$ and let $\mathbf{D}_{ji}$ be $d_{ij}^+$.

### 4.1.4 Structure-of-Influence Graph (SIG)

A Structure-of-Influence Graph (SIG) is a graph $G = (V, E)$ with one vertex per component and edges that represent influences. Edges may be undirected, directed, or even bidirectional, to indicate the delay(s) associated with this influence. This section explains how to construct a SIG.

Consider the $n \times n$ matrices $\mathbf{C}$ and $\mathbf{D}$. There is an edge between $i$ and $j$ if $\max(|\mathbf{C}_{ij}|, |\mathbf{C}_{ji}|) > \varepsilon$. Let $\alpha$ be the threshold for making an edge directed; the type of edge is determined as follows:

$$((|\mathbf{C}_{ij}| > \varepsilon) \Rightarrow (\mathbf{D}_{ij} > -\alpha)) \wedge ((|\mathbf{C}_{ji}| > \varepsilon) \Rightarrow (\mathbf{D}_{ji} < \alpha)) \qquad \Rightarrow i - j;$$

$$(|\mathbf{C}_{ij}| > \varepsilon) \wedge (\mathbf{D}_{ij} < -\alpha) \qquad \Rightarrow i \rightarrow j;$$

$$(|\mathbf{C}_{ji}| > \varepsilon) \wedge (\mathbf{D}_{ji} > \alpha) \qquad \Rightarrow i \leftarrow j;$$

$$(|\mathbf{C}_{ij}| > \varepsilon) \wedge (\mathbf{D}_{ij} < -\alpha) \wedge (|\mathbf{C}_{ji}| > \varepsilon) \wedge (\mathbf{D}_{ji} > \alpha) \qquad \Rightarrow i \leftrightarrow j.$$

The time complexity of our method on a system with $n$ components, given an algorithm to compute cross-correlation in time $O(m)$, is $O(n^2 m)$. For large systems, we may wish to compute only a subset of the SIG: all influences involving a set of $n' << n$ components. This is equivalent to filling in only specific rows and columns of $\mathbf{C}$ and $\mathbf{D}$ and requires time $O(nn'm)$.

Recall our example components $A$ and $B$. Using the cross-correlation of $A$ and $B$, shown in Figure 4.1, we apply the thresholds $\alpha = 20$ and $\varepsilon = 0.5$ and plot a SIG. Although $|\mathbf{C}_{ji}| < \varepsilon$, we have $|\mathbf{C}_{ij}| = 100 > \varepsilon$, so there will be an edge between $A$ and $B$. Furthermore, $\mathbf{D}_{ij} < -\alpha$, so the edge will be directed: $A \rightarrow B$. Figure 4.2 gives a SIG for a hypothetical system that includes $A$ and $B$ as components. In subsequent sections we discuss how the values of $\alpha$ and $\varepsilon$ can be chosen or set automatically.

### 4.1.5 Interpreting a SIG

An edge in a SIG represents a strong ($> \varepsilon$) influence between two components. The absence of an edge does not imply the absence of a shared influence, merely that the anomalies identified by the models are not strongly correlated—a different choice of models may yield a different graph. More specific interpretations arise from understanding particular models and underlying components.

A directed edge implies that an anomaly on the source component (tail of the arrow) tends to be followed shortly thereafter by an anomaly on the sink component (head of the arrow). Bidirectional edges mean that influence was observed in both directions, which may mean either that the influence truly flows in both directions or that it is unclear which directionality to assign (this situation can arise with periodic anomalies). An undirected edge means that, to within a threshold $\alpha$, the anomalies appear to occur simultaneously. This happens, for instance, when a mutually influential component is causing the anomalies. Such shared components sometimes introduce cliques into the SIG.

The model underlying each component could measure anything, from the message rate of a software application to the throughput variance of a network card. The edges of a SIG could surprise us by their existence (e.g., anomalies in fan speed are correlated with larger error terms in a scientific calculation), by their absence (e.g., anomalies in a disk subsystem do not influence the operating system), or by their directionality (e.g., a supercomputing workload whose anomalies precede those on the network router, rather than follow from them).

Structural properties of SIGs have useful interpretations. A clique, in particular, may draw attention to missing data: the influence shared by the components in the clique may be exerted by a component not represented explicitly in the SIG (although an unexpected clique may spur the user to include such a component).

When used for problem isolation, the most important piece of actionable information provided by our method is a concise description, in the form of graph edges, of which components seem to be involved. Further, the strength and directionality on those edges tell the order in which to investigate those components. In a system of black boxes, which is our model, this is the most any method can provide.

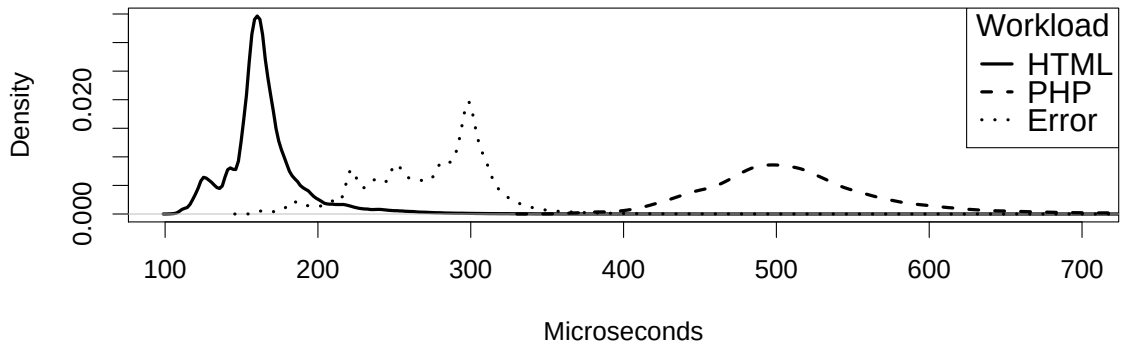



Figure 4.3: These `apache` workloads show that input semantics can affect output timing.

## 4.2 Controlled Experiments

In this section, we study the notion of influence and our method for computing it under a variety of adverse conditions: measurement noise, message loss, and tainted training data. We use simulation experiments on idealized systems consisting of linear chains of components. We use chains so the results of our simulations are easy to interpret; our method is not limited to chains, and our experiments with real systems in subsequent sections involve much more complex structure. Our goal is to thoroughly examine a specific question: Given a known channel of data and resource interactions through which influence could propagate, what is the strength ($\varepsilon$) of the influence inferred by our method? Our results show that, for many realistic circumstances, influence can propagate through long chains of components, and our method can detect it.

We first illustrate two common cases of influence flowing through components that arise in practice: semantic input differences and resource contention manifesting as downstream timing differences. Figure 4.3 shows that the web server `apache`, when given workloads consisting of error pages, html pages, or php pages, yields distinctive processing time distributions. That is, components can convert *semantic* information into *timing* information. Next, we show in Figure 4.4 how components' timing is affected by shared resources. When between one and four—otherwise independent—client-server pairs communicate using shared resources, they influence each other's

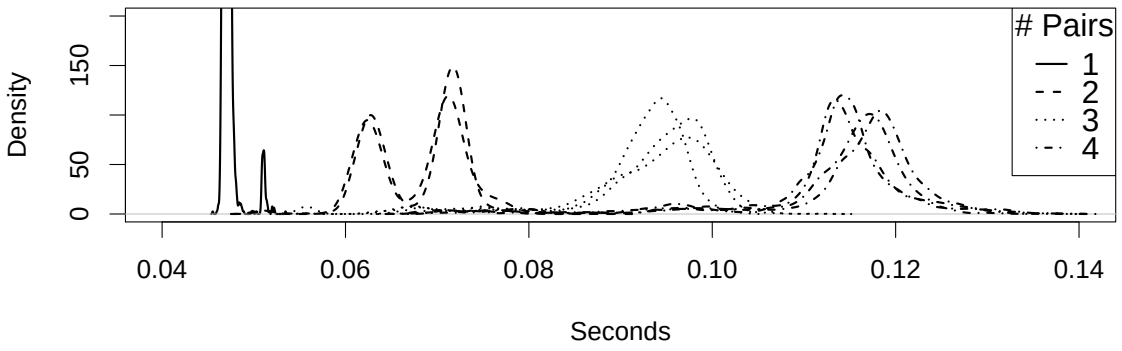



Figure 4.4: Client-server pairs can influence each other when they share a resource. As the number of otherwise-independent client-server pairs increases, thus increasing contention, the timing density distribution also changes.

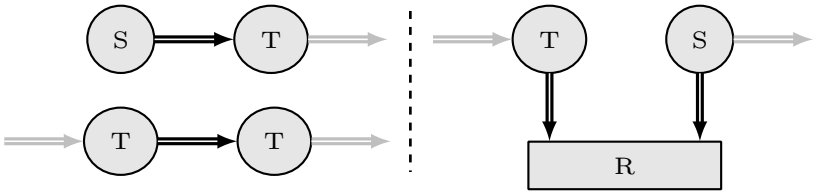



Figure 4.5: The three basic structures in our simulated systems, built from sources ("S"), tasks ("T"), and resources ("R").

timing: the distribution for a single client-server pair (far left) is very different from the distributions of the four simultaneous client-server pairs (far right).

### 4.2.1 System Components

The simulations use three types of components: *sources*, which generate data, *tasks*, which process data, and *resources*, which are required by sources and tasks to perform these actions. Pairs of sources and tasks, shown in Figure 4.5, can influence each other via direct communication or via competition over a shared resource. We study linear chains of such structures in which the first component is a source; that source, called the *head*, is designed to sometimes misbehave during the experiments and acts as the root cause that we wish to find. The only input to our method is a pair of timestamp vectors (one for the *head* of the chain and one for the *tail*) corresponding to message sending times. No information about simulation parameters or intermediate

components is provided.

Influence can flow over a direct source-to-task or task-to-task channel either by *timing* (anomalous input timing may cause anomalous output timing, as in a producer-consumer interaction), by *semantics* (tasks may take more or less time to process uncommon messages), or *both*. Influence can flow over a shared resource only through timing (e.g., the degree of contention may influence timing); we do not simulate implicit communication through shared memory.

### 4.2.2 Component Behavior

We characterize timing behavior of components by distributions of interarrival times, which is sufficient to compute anomaly signals (see Section 4.1.2). These experiments use Gaussian (normal) distributions. Let $\eta_x$ denote a normally distributed random variable with mean 1 and standard deviation $\sigma_x$. Fixing the mean makes the problem more difficult, because abnormal behavior does not result in consistently more or fewer messages (merely greater variance) and because anomalous behavior looks like measurement imprecision (noise).

A *source* generates the message `0` every $\eta_n$ seconds. A source may be *blocked* if any downstream component is not *ready* to receive the message, in which case it waits until all such components are ready, *sends* the message, and then waits $\eta_n$ seconds before trying to generate the next message. We consider three types of anomalous behavior at the source node: *timing* (generates a message every $\eta_a$ seconds), *semantics* (generates the message `1`), and *both*.

A *task* begins processing a message upon receipt, taking $\eta_0$ seconds for a `0` message and $\eta_1$ seconds for a `1` message. After processing, a task sends the same message to the next component downstream. If that component is not ready, the task is blocked. A task is ready when it is not blocked and not processing a message.

A *resource* receives and processes messages; it can simultaneously process up to $R$ messages for some capacity $R$. A resource requires $\eta_r$ seconds to process a message and is ready to receive whenever it is processing fewer than $R$ messages. Resources service simultaneous requests in a random order.

When the head (the source) or tail of the chain sends a message, as described above, it records the time at which the message was sent; our method computes the influence given only this pair of timestamp lists. While real systems may exhibit more complex behavior, these simple rules are enough to capture different classes of inputs (`0` vs. `1` messages), resource contention, and potential timing dependencies.

### 4.2.3 Methodology

Each experiment, resulting in a single influence value $\varepsilon$, involves two independent simulations of a chain over a time period long enough for the head to send 10,000 messages. The first simulation yields a trace that is used for training (a behavior baseline), and the second, a monitoring trace, is used to build the SIG. Except where otherwise indicated, the training trace does not contain anomalous behavior, and the monitoring trace contains a contiguous period of anomalous behavior lasting 5% of the trace (500 messages).

Resources have exactly two connected components, each with a normal average message sending rate of 1 per second, so every resource has a capacity of $R = 2$. That is, there should be little contention during normal operation. The number of resources is denoted by `#R=?`; resources are evenly distributed along the chain. Except where otherwise noted, $\sigma_n = \sigma_r = \sigma_0 = 0.01$ and $\sigma_a = \sigma_1 = 0.1$. For the component model, the histogram bin size is $h = 0.01$ seconds (set automatically) and the window size is $w = 500$ samples (chosen to match the anomalous period).

### 4.2.4 Experiments

**Baseline:** We compute an influence strength baseline for each simulation that represents the expected correlation of anomaly signals if the head and tail were independent. For all experiments in this section, that baseline ranges from 0.06 to 0.1; the average for each set of experiments is plotted in the figures as a dashed line. Thus, any $\varepsilon$ value above 0.1 can be considered statistically significant. This is comparable to the edge threshold of $\varepsilon = 0.15$ that we use when considering real systems (see Sections 4.3 and 4.4).

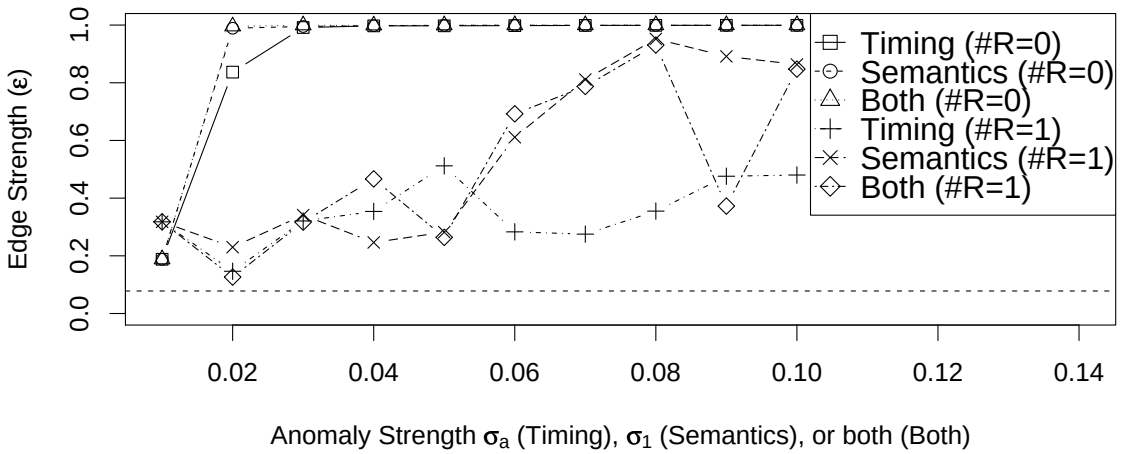



Figure 4.6: Behavior of the basic components: a single task (#R=0) and a single resource (#R=1), each with a driving source head.

**Basic Components:** Figure 4.6 shows the strength of influence across the basic simulation components of Figure 4.5 for varying anomaly strengths. Tasks propagate both timing and semantic influence, while resources only propagate timing. Tasks change semantic influence into timing influence, however, which resources can then propagate. Over-provisioned resources do not propagate influence; resources with adequate capacity, which we simulate, propagate timing influence. Note that we detect influence even when anomalous behavior looks similar to normal behavior: even during normal operation there is variation in component behavior, and these variations may be correlated between components.

**Length and Composition:** When there is more than one component, we find that influence generally fades with increasing chain length, but remains detectable (see Figure 4.7). When there are no resources, however, message semantics are passed all the way to the tail and the influence is undiminished. For the rest of the section, chains contain six components; this length is long enough to exhibit interesting properties and is comparable to the diameter of the autonomous vehicle graphs in Section 4.3.

**Signal Noise:** Our method is robust against noisy data. As we add more and more Gaussian noise to the timing measurements, it obscures some of the influence of anomalous behavior but does not mask it entirely. This is true when noise is added

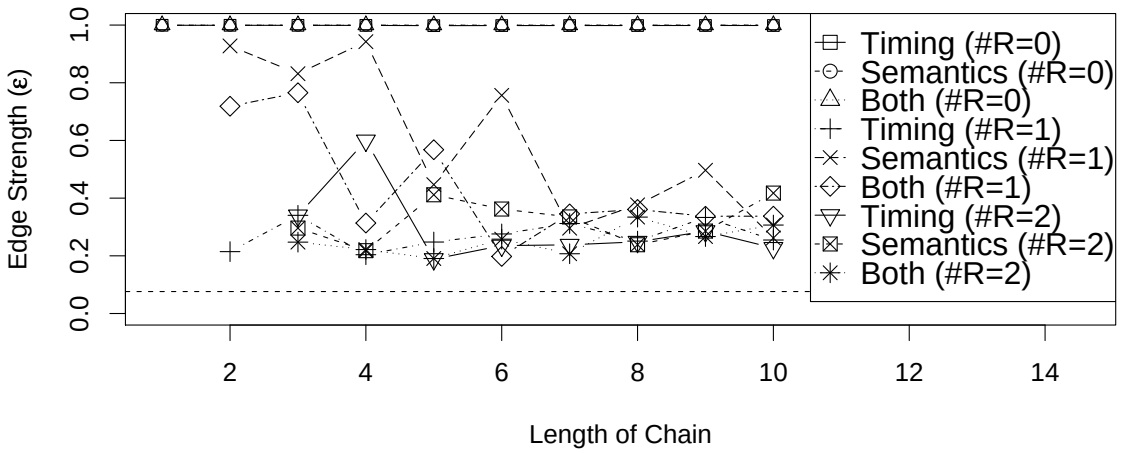



Figure 4.7: Contention carries timing influence across resources and tasks pass along semantic influence, even down long chains.

to the resulting measurements (measurement imprecision, as in Figure 4.8) or to the components (omitted for space, but similar to Figure 4.8). Note that even "normal" timing variations at the head can influence timing at the tail.

**Message Loss:** For our timing model, message loss is simply another form of noise that tends to introduce outliers. For example, if a component output messages at $t_1$, $t_2$, and $t_3$, but the second measurement is lost, our timing distribution will erroneously include the value $t_3 - t_1$, which will be twice as large, on average, as most of the other measurements. To make our job more difficult, we simulate the case when our training data has no lost messages but the monitoring data does. Figure 4.9 shows that our statistical methods are not strongly sensitive to missing data.

**Tainted Training:** The problem of good training data exists for every anomaly-based method. Figure 4.10 shows that, as the fraction of training data that includes anomalous behavior increases, influence remains easily detectable. Tainting does not tend to introduce new correlations; existing correlations may appear less significant, as in the middle line. Training data need only be statistically representative, so it can include unusual periods (like startup) or bugs.

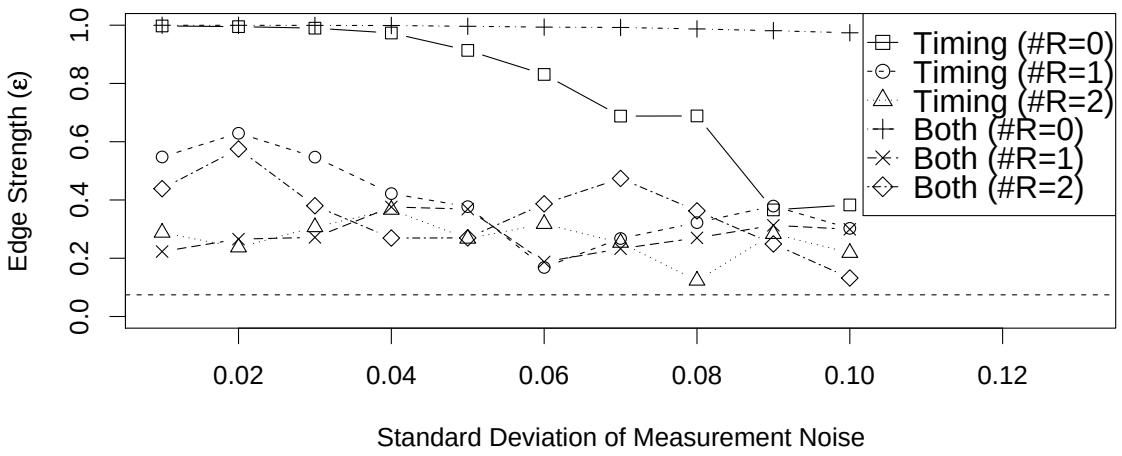



Figure 4.8: Our method degrades gracefully when timing measurements are noisy.

These experiments show that influence propagates through systems in a measurable way and that our method can detect this influence under a variety of circumstances. Although the simulations consider a restricted class of systems, the systems we study in Sections 4.3 and 4.4 contain far more complex structure, including asynchronous communication through shared memory, high degrees of network fan-in and fan-out, loops and cycles, and potentially multiple sources of anomalous behavior.

## 4.3 Stanley and Junior

DARPA launched the Grand Challenge in 2003 to stimulate autonomous vehicle research. The winner of the Grand Challenge was a diesel-powered Volkswagen Touareg R5 named Stanley, an autonomous vehicle developed at Stanford University [70]. Stanford's entry in the successive contest, a modified 2006 Volkswagen Passat wagon named Junior, placed second in the Urban Challenge [36].

Many of the autonomous vehicles' components run in tight loops that output log messages at each iteration. Deviations from normal timing behavior are rare, but, more importantly, we expect the anomalies to correspond with semantically abnormal situations. For example, if the route-planning software takes unusually long to plot a path, the vehicle may be in a rare driving situation (e.g., a 4-way stop where the

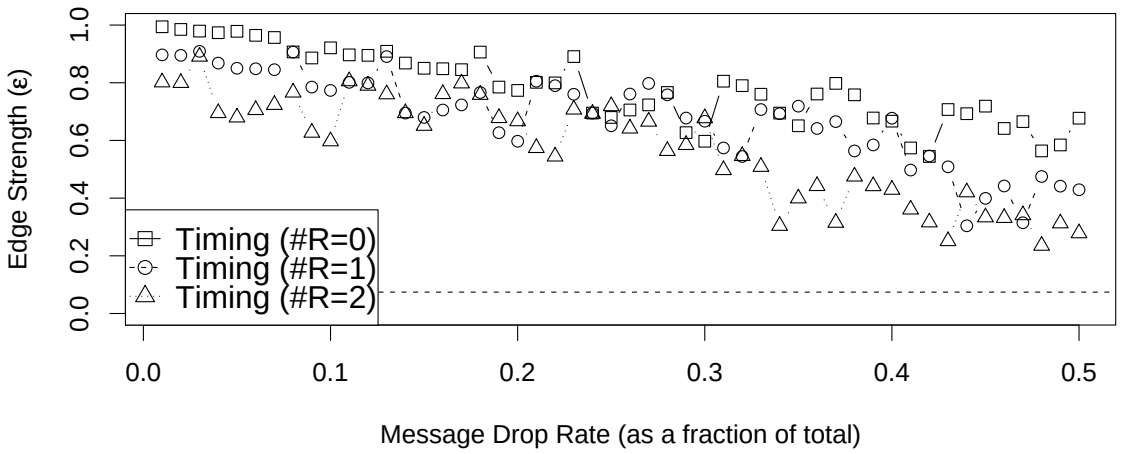



Figure 4.9: Our method is robust against uniform message loss, even at rates of 50%.

driver with right-of-way is not proceeding).

### 4.3.1 Stanley's Bug

During the Grand Challenge race, Stanley suffered from a bug that manifested itself as unexplained swerving behavior. That is, the vehicle would occasionally veer around a nonexistent obstacle. According to the Stanford Racing Team, "as a result of these errors, Stanley slowed down a number of times between Miles 22 and 35" [70]. The bug forced Stanley off the road on one occasion, nearly losing the race by disqualification. We explain this bug in more detail in Section 4.3.6, but, for the time being, let us suppose that all we know is that we were surprised by Stanley's behavior between Miles 22 and 35 of the race and that we would like to use the method described in this thesis to find an explanation.

### 4.3.2 Experiments

During the Grand Challenge and Urban Challenge races, each vehicle was configured to record inter-process communication (IPC) between software components, including the sensors. These messages were sent to a central logging server and written to disk. Only a subset of the components generated logs. The messages indicate their source,

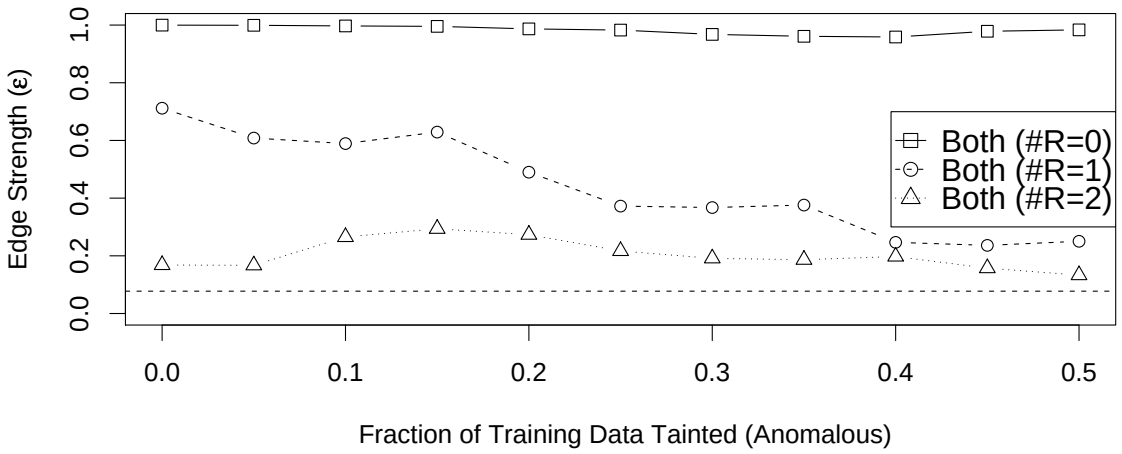



Figure 4.10: Our ability to detect influence does not depend on collecting clean training data.

but not their destination; the absence of such information means that most previous work would be inapplicable to this data set. We sample the anomaly signals at intervals of 0.04 seconds; this sampling interval was set (automatically) to be the smallest non-zero interarrival time on any single component on Stanley.

The log format is quite regular, with the following canonical format representing all components (with exceptions noted below). The ellipsis indicates sundry text:

```
[component] [payload] [utime] [...] [logger time]
```

The `component` field does not, strictly, denote the source of the message; it may, for example, be a message destined for that component. In this work, however, we treat each such named component as a distinct entity. The absence of a "destination" field means that methods requiring message paths are inapplicable.

Starting with the raw race logs, we make a few modifications. Our method applies most directly to components that produce measurements throughout the span of the logs, so we remove components that were only active during initialization or for only part of the race. Some of the excluded "components" were status messages, however, generated by a single component that also generated regular messages under a different name. For example, we removed `PASSAT_STATUS` but retained `PASSAT_OUTPUT2`.

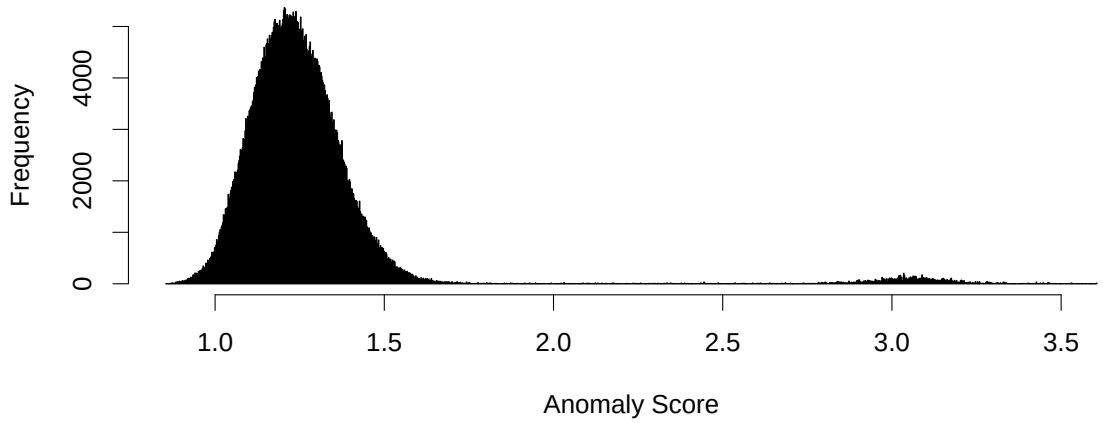



Figure 4.11: Anomaly signal distribution for Stanley's `GPS_POS` component.

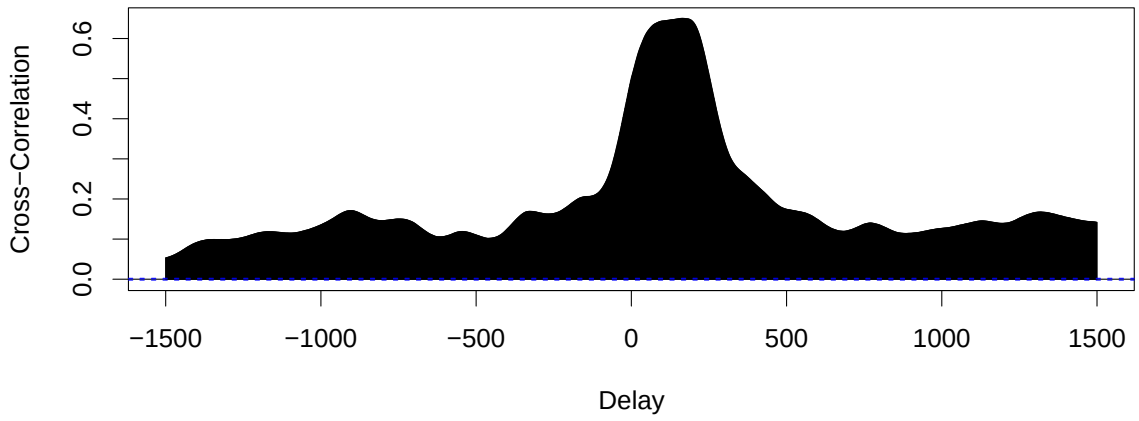



Figure 4.12: Cross-correlation of Stanley's `PLANNER_TRAJ` and `LASER1` components.

We also trim the logs slightly at both ends to the points where all remaining components are active.

Computing a SIG requires only two parameters, $\varepsilon$ and $\alpha$, neither of which need to be fixed *a priori*; adjusting them to explore the impact on the resulting graph can be informative (e.g., if an entire clique becomes disconnected due to a small decrease in $\varepsilon$, we know that the shared influence has roughly the same impact on all members of the clique). For the component model, the histogram bin size(s) $h$ is set automatically using Sturges' formula [67], and we use a recent window size of $w = 100$ samples; our results are not sensitive to this choice.

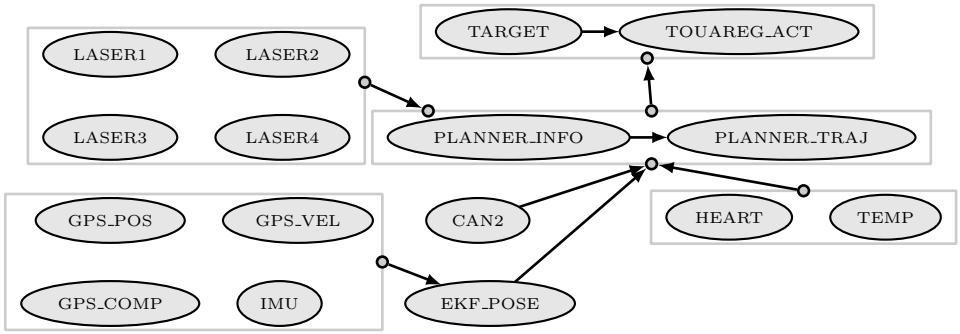



Figure 4.13: Known dependency structure of Stanley, including only logged components.

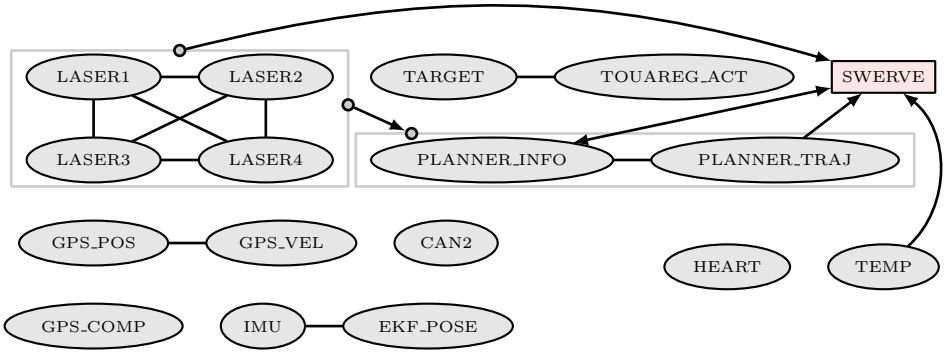



Figure 4.14: The automatically generated SIG for Stanley, with $\varepsilon = 0.15$ and $\alpha = 90$. The special `SWERVE` component is explained in Section 4.3.6.

### 4.3.3 Anomaly Signals

For each component, we use the timing model and computations described in Section 4.1.2 to generate an anomaly signal. A "good" anomaly signal has low variance when measuring a system under typical conditions, in accordance with its semantics (usual behavior is not surprising behavior). Often, the vehicle components generate normally distributed or exponentially distributed anomaly scores. Sometimes, as in Figure 4.11, the anomaly scores are bimodal, with one cluster around typical behavior and another cluster around anomalous behavior.

### 4.3.4 Cross-correlation

We proceed by computing the cross-correlation between all pairs of components within each car using a discrete version of Equation 5.1.1. When two components do not share an influence, the cross-correlation tends to be flat. This can also happen when two components share an influence but there is no consistent delay associated with it. When there is a shared influence, we see a peak or valley in the cross-correlation function. The more pronounced the extrema, the stronger the inferred influence. Figure 4.12 gives the cross-correlation between Stanley's `PLANNER_TRAJ` and `LASER1` components. We see a peak whose magnitude (correlation) exceeds 0.6, which is relatively large for this system. We can already conclude that `PLANNER_TRAJ` and `LASER1` likely share an influence. The strong correlation at a small positive lag means the `LASER1` anomalies tend to precede those on `PLANNER_TRAJ`.

In addition to the magnitude of the correlation, we can learn from the location of an extremum on the delay axis. Here, we see that it occurs at a delay of, roughly, 100–200 samples (at a 0.04-second sampling interval, the delay is around 4–8 seconds). In this case, we are looking at the result of computing $(\texttt{PLANNER\_TRAJ} \star \texttt{LASER1})(t)$, so the interpretation is that anomalies on `PLANNER_TRAJ` tend to occur between 4–8 seconds *after* those on `LASER1`. More important than the value, however, is the direction: the laser anomalies precede the planner software anomalies. When isolating the bug mentioned in Section 4.3.1, this turns out to be an important piece of information.

### 4.3.5 SIGs

Using the method described in Section 4.1.4, we distill the cross-correlation matrices into SIGs. We compute a statistical baseline for Stanley, similar to Section 4.2.4, of just under 0.15. For Stanley, a SIG with $\varepsilon = 0.15$ and $\alpha = 90$ is shown in Figure 4.14; the hand-generated software dependency diagram for Stanley is shown in Figure 4.13. As a notational shorthand to reduce graph clutter, we introduce grey boxes around sets of vertices. An arrow originating from a box denotes a similar arrow originating from every vertex inside the box; an arrow terminating at a box indicates such an arrow terminating at every enclosed vertex. Consequently, the

directed arrow in Stanley's SIG from the box containing LASER* to the box containing PLANNER* indicates that each laser component shares a time-delayed influence with each planning software component. We explain the SWERVE component, plotted as a red rectangle, in Section 4.3.6. Most of the strongest influences do not map to stated dependencies, meaning the dependency diagram has obscured important interactions that the SIG reveals.

The edges in the dependency diagram indicate intended communication patterns, rather than functional dependencies. In fact, Stanley had five laser sensors, not four, but one broke shortly before the race. The downstream components were clearly not dependent on that laser, in the sense that they continued working. If another laser malfunctioned, would it affect the behavior of the vehicle? The dependency diagram is unhelpful, but in Section 4.3.6 we show how to query a SIG to elucidate this kind of influence.

Even in an offline context, SIGs are dynamic. By using only a subset of the data, such as from a particular window of time, we can see the structure of influence particular to that period. Furthermore, we can consider a series of such periods to examine changes over time. A SIG for Junior showing influence during the second race mission, relative to the first is plotted in Figure 4.15. Dashed grey means the edge is gone, thicker edges are new, and an open arrowhead means the arrow changed. Disconnected components are omitted. Although the component models use the entire log as training data, we generate this graph using only data from the first and second thirds of the Urban Challenge (called "missions"), with edges marked to denote changes in structure. Notice that many components are disconnected in this SIG, and thus omitted from the plots, as a consequence of the value of $\varepsilon$. As this parameter increases, edges vanish; as it tends to zero, the graph becomes fully connected.

One notable change is the new influence between RADAR5 and PASSAT_EXTOUTPUT2. Further examination of the data shows several anomalies at the radar components; a lower value of $\varepsilon$ would have shown the radars in a clique, but RADAR5 exhibited the most pronounced effect. Many of the edges that disappeared did so because anomalies during the first mission did not manifest in the second. Studying changes in influence

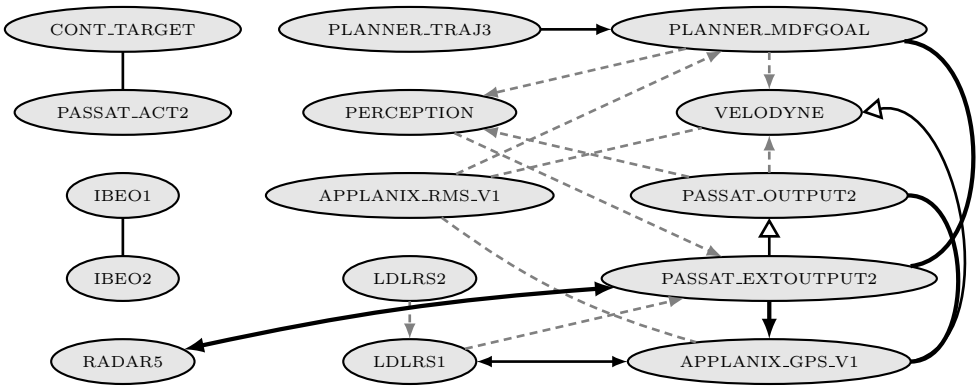



Figure 4.15: Dynamic changes in Junior's SIG, with $\varepsilon = 0.15$ and $\alpha = 90$.

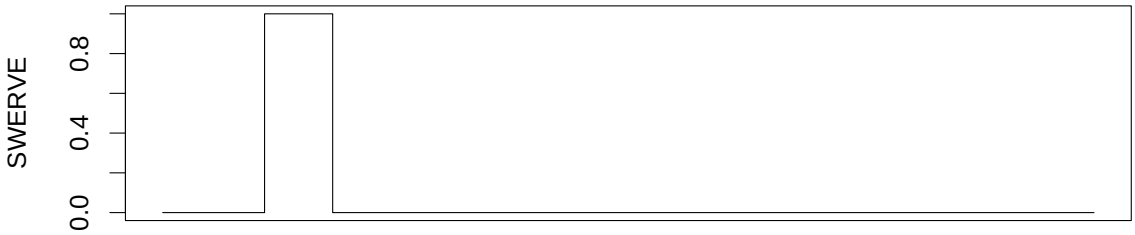



Figure 4.16: Anomaly signal for a synthetic `SWERVE` component, taking a nonzero value only around the time surprising behavior (swerving) was observed.

structure over time enables us to discover such dynamic behaviors.

For the vehicles, physical connectivity does not change during a race, so any changes are a consequence of which influences we happened to observe during that period. If we only had access to the latter half of the Grand Challenge logs, for instance, it would have been more difficult to diagnose the swerving bug because there was little swerving behavior to observe. For larger systems and longer time periods, changes in the SIG may correspond with upgrades or physical modifications.

### 4.3.6 Swerving Bug

Starting only with the logs from the Grand Challenge race and the knowledge that Stanley exhibited strange behavior between Miles 22 and 35, we show how a simple application of our method points directly at the responsible components.

First, we construct a synthetic component called SWERVE, shown in Figure 4.16, whose anomaly signal is nonzero only for a single period that includes the time the majority of surprising behavior (swerving) was observed. We could refine this signal to indicate more precisely when the bug manifested itself, thus increasing the strength of the correlation, but the surprising result is that, even with this sloppy specification of the bug, our statistical method is still able to implicate the source of the problem.

Second, we update Stanley's SIG as though SWERVE were just another component. The result is the graph in Figure 4.14. Consider the correlation values for the seven components with which SWERVE seems to share an influence: all four laser sensors, the two planning software components, and the temperature sensor.

The temperature sensor is actually anti-correlated with SWERVE. This spurious correlation is the price we pay for our sloppy synthetic anomaly signal; it occurs because the SWERVE anomaly signal is (carelessly) non-zero only near the beginning of the race while the TEMP anomaly signal, it turns out, is increasing over the course of the race.

Now there are six components that seem to be related to the swerving, but the SIG highlights two observations that narrow it down further: (i) directed arrows from the lasers to the planner software mean that the laser anomalies preceded the planner anomalies and (ii) the four lasers form a clique, indicating that there may be another component, influencing them all, that we are not modeling. (Recall the discussion in Section 4.1.5.) Observation (i) is evident from the SIG and Figure 4.12 confirmed that the planner typically lagged behind the lasers' misbehavior by several seconds. Observation (ii) suggests that a component shared by the lasers, rather than the lasers themselves, may be at fault.

According to the Stanford Racing Team, these observations would have been sufficient to quickly diagnose the swerving bug. At the time, without access to a SIG and working from the dependency diagrams, it took them two months to chase down the root cause. This is a natural consequence of how a dependency diagram is used: start at some component near the bottom of the graph, verify that the code is correct, move up to the parent(s), and repeat. All of the software was working correctly, however, and until they realized there were anomalies at the input end (the opposite

end from the swerving) there was little success in making a diagnosis. The SIG would have told them to bypass the rest of the system and look at some component shared by the lasers, which was the true cause.

The bug, which turned out to be a non-deterministic, implicit timing dependency triggered by a buffer shared by the laser sensors, is still not fully understood. However, the information our method provides was sufficient for the Racing Team to avoid these issues in Junior. Indeed, this is evident from Junior's SIG, Figure 4.15, where the lasers are not influencing the system as they were in Stanley. More information on the swerving bug can be found in the journal paper concerning Stanley [70].

## 4.4 Thunderbird Supercomputer

We briefly describe the use of SIGs to localize a non-performance bug in a non-embedded system (the Thunderbird supercomputer from Chapter 2) of significantly larger scale ($n = 9024$) using a component model based on the frequency of terms in log messages (the Nodeinfo algorithm of Chapter 3) instead of timing. Except for the different anomaly signal, the construction and use of the SIGs is identical to the case study in Section 4.3; we focus only on the new aspects of this study.

Say that Thunderbird's administrator notices that a particular node generates the following message and would like to better understand it: `kernel: Losing some ticks... checking if CPU frequency changed.` For tightly integrated systems like Thunderbird, static dependency diagrams are dense with irrelevant edges; meanwhile, the logs do not contain the information about component interactions or communication patterns necessary for computing dynamic dependencies.

Instead, we show how a simple application of our SIG method leads to insight about the bug. Using Nodeinfo, which incorporates no system-specific knowledge, we first compute anomaly signals for all components. As with Stanley's swerving bug (see Section 4.3.1), we can easily synthesize another anomaly signal for each node, nonzero only when it generates this "CPU error".

The resulting SIG contains clusters of components (both synthetic and normal) that yield a surprising fact: when one node generates the CPU error, other nodes

tend to generate both the CPU error and other rare messages. Furthermore, the administrator can quickly see that these clusters correspond to job scheduling groups, suggesting that a particular workload may be triggering the bug. Indeed, it turns out that there was a bug in the Linux kernel that would cause it to skip interrupts under heavy network activity, leading the kernel to erroneously believe that the clock frequency had changed and to generate the misleading CPU error. The SIG method shows immediately that the error shares an influence with other nodes in the same job scheduling group; an insight that helps both isolate the cause and rule out other avenues of investigation.

## 4.5 Contributions

In this chapter, we propose using *influence* to characterize the interactions among components in a system and present a method for constructing Structure-of-Influence Graphs (SIGs) that model the strength and temporal ordering of influence. We abstract components as *anomaly signals*, which enables noise-robust modeling of heterogeneous systems. Our simulation experiments and case studies with two autonomous vehicles and a production supercomputer show the benefits of using influence over traditional dependencies and demonstrate how an understanding of influences can equip users to better identify the sources of problems.

# Chapter 5

# Query Language

A large system may have many thousands of components that influence each other, of which only a small subset are relevant for diagnosing a particular problem. This chapter builds a query language on top of our influence method that allows users to constrain the computation of influence to those components that are likely to be relevant. A query can be as simple as a description of the times when a particular problem was observed or a list of suspicious components; our system QI (for Querying Influence; pronounced "CHĒ") uses this information to constrain the problem and produce a smaller, more useful Structure-of-Influence Graph, more quickly. This extension to SIGs makes them more usable and scalable.

For example, the administrator might specify that surprising behavior was observed around the times a user's jobs failed. Using that clue alone, QI determines what other components deviated from normal behavior around those times and generates a SIG to summarize the correlations (see Section 5.2). For example, QI might generate the hypothetical SIG in Figure 5.1, which shows a component called `fan7` sharing a strong influence with `disk32`, which in turn shares a strong influence with `job failures` (a node representing the problem). The directed arrows imply a temporal ordering, which in practice often indicates causality. Furthermore, `fan7` also shares an influence with `fan6`, possibly alerting the administrator to investigate whether some common cause (perhaps in another component that is not instrumented and produces no logs) is making the fans misbehave, which in turn is related to disk misbehavior,

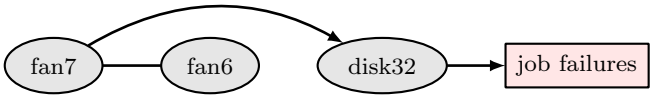



Figure 5.1: An example SIG showing a chain of influence related to the job failures.

which in turn is likely related to the job failures.

QI does not require modifications or perturbations to the system, access to source code, or even knowledge of all the components in the system or their dependencies on one another. Our assumptions are considerably weaker than most previous work and they reflect, in our experience, the reality faced by administrators when they must diagnose a problem. The answers QI provides are limited by these contraints: a passive, black-box technique can, at best, suggest the components and interactions that seem statistically most likely to be involved with a problem. The main advantage is that, because of the weak assumptions, such a technique can leverage all of the available information. This is precisely what our method provides, and it does so in a way that is computationally efficient and applicable to a wide variety of systems.

In this chapter, we present QI's query language (see Section 5.1) and our implementation of an infrastructure for efficiently computing these queries (see Section 5.2). Following the lead of previous work in this area [32, 61], we primarily evaluate our method with case studies; Section 6.3 shows how we can use QI to help isolate a variety of interesting problems, including a non-deterministic timing bug and an operating system kernel bug triggered by particular workloads.

## 5.1 The Query Language

A system is composed of a set of *components*; we assume we are given some subset of all system components that produce logs with time-stamped measurements. In QI, a component is represented by a time-series vector describing its behavior; this is the component's *anomaly signal*.

Consider the hypothetical components `fan7` and `disk32` from Figure 5.1, whose anomaly signals are plotted in Figure 5.2. These two signals have similar structure,

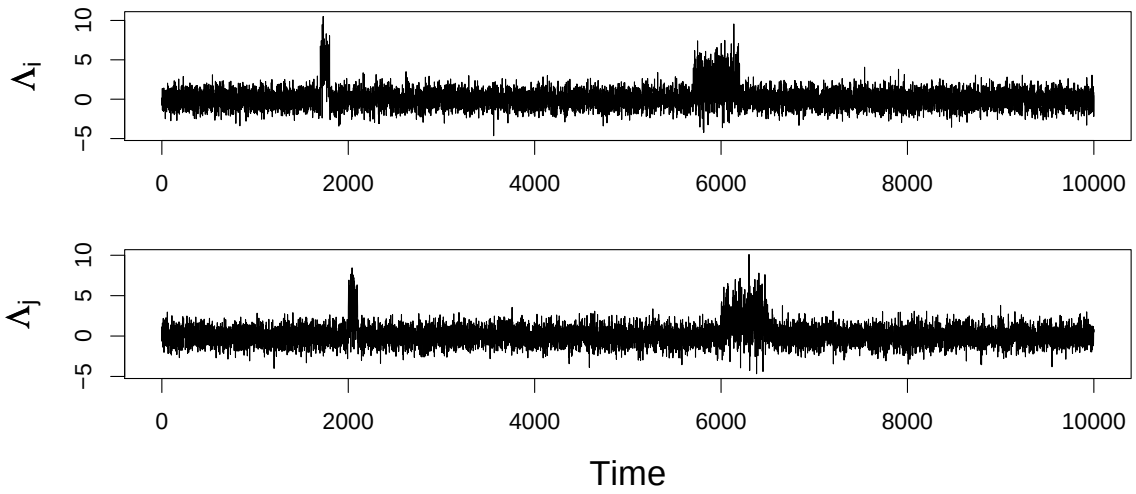



Figure 5.2: Anomaly signals for the hypothetical components `fan7` ($\Lambda_i$) and `disk32` ($\Lambda_j$). By inspection, the signals look similar; our method mathematically describes this similarity.

especially around times 2000 and 6000. This similarity is what many system administrators search for manually and is what our method extracts and summarizes automatically and at scale.

Recall that if the anomaly signal of one component is correlated with the anomaly signal of another component, we say that these components share an *influence*. A Structure-of-Influence Graph (SIG) is a graph in which the nodes are components and the edges summarize the strength and directionality of shared influence between components. The SIG in Figure 5.1 shows a directed arrow from `fan7` to `disk32`, which means that surprising behavior on `fan7` tends to be followed a short time later by surprising behavior on `disk32`—that is, the probability that `disk32` will show surprising behavior increases in a short period after `fan7` shows surprising behavior. Section 5.1.1 describes these computations. Furthermore, a query may specify new components that are built from existing components (e.g., a group of components) or additional information (e.g., a time range); Section 5.1.2 explains these *synthetic components* using several examples.

The user may wish to view only a fragment of the complete SIG, consisting of some subset of the components and edges; such components are *in focus*. A *query*

specifies what components are in focus by naming them, and QI will compute all pairwise relationships between components in focus. Additionally, the user can specify a set of components for which not all pairs should be computed, called the *periphery*. QI will examine all pairs within the focus and between each component in the focus and each in the periphery, but will not examine the relationships between any two components in the periphery. This is useful when, for example, we want to know what focal components share the strongest influence with a set of peripheral components, but we don't yet care how those peripheral components, in turn, influence each other. We provide a formal description of queries in Section 5.1.3.

### 5.1.1 Query Mathematics

QI computes cross-correlations between pairs of components and stores characteristics of these results in a pair of matrices: one for *correlation* magnitudes and one for associated *delays*. The resulting SIG summarizes these correlations and delays in the form of edges between components. We explain these computations using the example from Section 6, by describing how QI infers the directed edge from `fan7` to `disk32`.

To determine whether unusual behavior on `fan7` (represented by the anomaly signal $\Lambda_i$) correlates in time with unusual behavior on `disk32` ($\Lambda_j$) we compute the normalized dot product; the product will be larger if the anomaly signals tend to line up. This alignment may occur with some delay, however, so we use cross-correlation to check for correlation when the signals are delayed relative to each other:

$$(\Lambda_i \star \Lambda_j)(t) \equiv \int_{-\infty}^{\infty} \frac{[\Lambda_i(\tau) - \mu_i][\Lambda_j(t+\tau) - \mu_j]}{\sigma_i \sigma_j} d\tau,$$

where $\mu_i$ and $\sigma_i$ are the mean and standard deviation of $\Lambda_i$, respectively. Figure 5.3 shows this function for `fan7` and `disk32`. There is a peak at delay $t = d$ with correlation strength $c$; we now describe how such salient features are summarized.

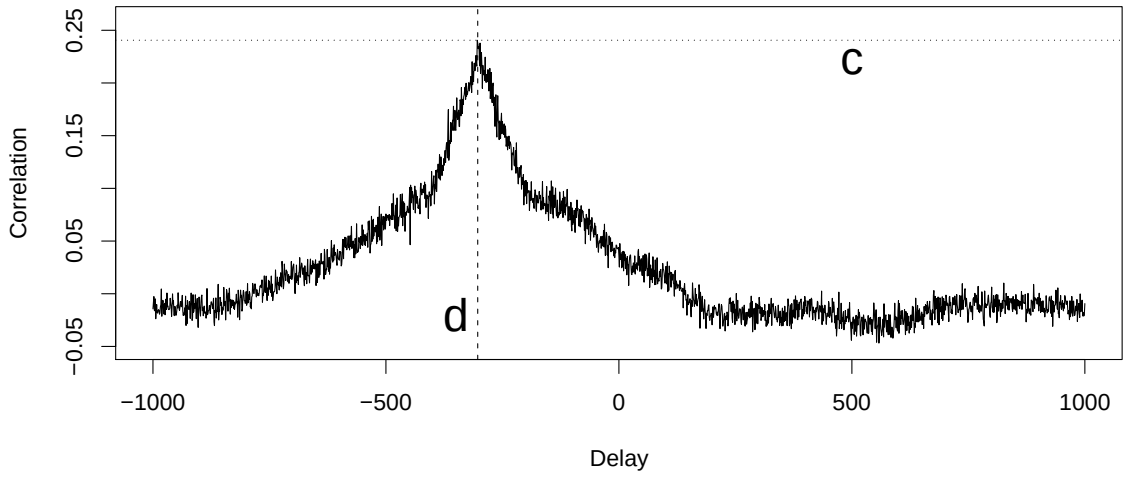



Figure 5.3: Cross-correlation between the anomaly signals of `fan7` and `disk32`. The peak at delay $d$ has height (correlation) $c$, indicating that surprising behavior on `fan7` tends to be followed by surprising behavior on `disk32` with a lag of $d$.

Let $d_{ij}^-$ and $d_{ij}^+$ be the offsets closest to zero, on either side, where the cross-correlation function is most extreme:

$$\begin{aligned} d_{ij}^- &= \max(\text{argmax}_{t \le 0}(|(\Lambda_i \star \Lambda_j)(t)|)) \text{ and} \\ d_{ij}^+ &= \min(\text{argmax}_{t \ge 0}(|(\Lambda_i \star \Lambda_j)(t)|)), \end{aligned}$$

where $\text{argmax}_t f(t)$ is the set of $t$-values at which $f(t)$ is maximal. Intuitively, $d_{ij}^-$ and $d_{ij}^+$ capture the most common amount of time that elapses between surprising behavior on `fan7` and surprising behavior on `disk32`. Referring again to Figure 5.3, the peak is at $d < 0$, so $d_{ij}^- = d$. Next, let $c_{ij}^-$ and $c_{ij}^+$ be the correlations observed at those extrema:

$$\begin{aligned} c_{ij}^- &= (\Lambda_i \star \Lambda_j)(d_{ij}^-) \text{ and} \\ c_{ij}^+ &= (\Lambda_i \star \Lambda_j)(d_{ij}^+). \end{aligned}$$

Intuitively, $c_{ij}^-$ and $c_{ij}^+$ represent how strongly the behaviors of `fan7` and `disk32` are correlated. (From Figure 5.3, $c_{ij}^- = c$.)

We record these summary values of cross-correlations in the correlation matrix $\mathbf{C}$ and delay matrix $\mathbf{D}$. Let entry $\mathbf{C}_{ij}$ be $c_{ij}^-$ and let $\mathbf{C}_{ji}$ be $c_{ij}^+$. (Notice that $c_{ij}^+ = c_{ji}^-$.) Similarly, let entry $\mathbf{D}_{ij}$ be $d_{ij}^-$ and let $\mathbf{D}_{ji}$ be $d_{ij}^+$.

An edge appears between `fan7` and `disk32` in the SIG because their unusual behavior was sufficiently strongly correlated; e.g., $\max(\mathbf{C}_{ij}, \mathbf{C}_{ji}) = c > \varepsilon$, for some threshold $\varepsilon$ specified—implicitly or explicitly—by the query. The edge is directed because the corresponding delay $d$ lies outside of some noise margin $\alpha$. For instance, say $c > \varepsilon > \mathbf{C}_{ji}$. Because $|d| > \alpha$, the edge is directed with the tail at `fan7` and the head at `disk32`.

Clock skew between components could result in an incorrect delay inference. If the skew is known, the solution is simply to time-shift the data, accordingly; otherwise, the amount of skew may be inferred by looking at the delay between two components thought to behave in unison. Clock skew was not an issue on any of the systems we studied (see Section 5.3).

In addition to specifying the components and edges to include in the SIG, a query may also create new components by combining the anomaly signals for some existing components $c_1, \ldots, c_n$ into a new anomaly signal $f(c_1, \ldots, c_n)$ for some function $f$. For example, the behavior of a collection of homogeneous components (e.g., all the compute nodes or all the I/O nodes) can be represented by the average of the anomaly signals of the group's components.

### 5.1.2 Query Examples

In this section, we provide some example queries to build an intuition for what computations QI performs.

#### Metacomponents

A set of components can be grouped and named by creating a metacomponent. The anomaly signal for a metacomponent is the average of the signals of its constituent components. For example, we could create a metacomponent for all the nodes in a particular rack of a supercomputer (topological group) or all the sensors of a particular

type in an embedded system (functional group). Our current implementation specifies metacomponents using regular expressions in a configuration file.

Say that our system contains metacomponents for each rack of nodes (named `rack1`, `rack2`, etc.) and we are trying to understand strange behavior observed on component `r58node7`. It may be expensive to compute the pair-wise relationships between `r58node7` and all other nodes. Instead, we can start with the metacomponents:

```
graph temp top=5 periph=meta r58node7
```

This query will construct a SIG and write it to a file called "temp.dot" in the DOT language. (In subsequent examples, we omit `graph` and the filename.) In this query, only `r58node7` is in focus; the keyword `meta` after the `periph` parameter indicates that the set of metacomponents are in the periphery. QI will compute all pairs (`r58node7`, `n`), where `n` is a metacomponent. QI supports a variety of parameters for specifying how the resulting SIG should be visualized. In this query, the `top` parameter dictates that only the strongest 5 edges will be included in the graph.

Say that nodes in `rack3` are named with the prefix `r3` and that `rack3` exhibits the strongest shared influence with `r58node7`. We can refine the search to examine all the components in `rack3`:

```
r58node7  ^r3
```

The problem node and all components matching the regular expression are in focus. This query inherits the value of the `top` parameter from the previous query, so we omit it. The resulting graph directs the search for a problem cause toward a small set of nodes; in Section 5.4.1, we use QI in a similar way to understand job failures on a supercomputer.

### Binary Components

A query can create a new component whose anomaly signal is high exclusively during a particular time interval `a:b`. An interval may consist of several sub-intervals: `a:b,c:d,...`. We use range binary components in Section 5.4.1 to identify a data corruption problem and in Section 5.4.6 to isolate an implicit timing dependency bug in an autonomous vehicle.

Consider the following example query:

```
arrow=10s edge=0.25 47:50 all
```

The range portion of the query, `47:50`, will create a component whose anomaly signal is non-zero only between time 47 and time 50, inclusive, and add it to the system. A binary component only needs a name if we intend to use it in a subsequent query, and it is unnecessary to specify a magnitude because our method implicitly normalizes signals when it computes influence. The keyword `all` is shorthand for every component currently in the system. The `edge` parameter sets the $\varepsilon$ threshold (see Section 5.1.1) so that only correlations stronger than 0.25 appear as edges in the SIG; the `arrow` parameter sets the $\alpha$ threshold for making a graph edge directed, so if the absolute delay associated with some correlation is greater than 10 seconds, the edge will be directed.

A binary component can also be constructed using some predicate that QI can evaluate. For several supercomputer examples in Section 6.3, we construct binary components whose anomaly signals are high exclusively in intervals where the corresponding logs match a regular expression.

### Masked Components

A binary component, like a range, can be applied to another component as a *mask*. Anomaly signal values within the specified sub-intervals are left alone; values outside the sub-intervals are set to be the average value of the signal within the sub-intervals. For example, if the (very short) anomaly signal is $\Lambda_j = (0, 1, 0, 1, 0, 1)$, then the query term `j'=j{1:2,5:6}` results in the following signal for the new component `j'`: $\Lambda_{j'} = (0, 1, 0.5, 0.5, 0, 1)$.

Masks are useful for removing the influence from anomalous behavior that is already understood (e.g., one might choose to ignore daily reboots) and for attribution (determining what parts of the anomaly signal are contributing to an observed correlation; see Section 5.4.3). For example, say that we understand the anomalies on `fan7` and wish to mask them. We can construct the binary component on the top in Figure 5.4, apply it to `fan7` (Figure 5.2, top), and get the masked component on the bottom in Figure 5.4.

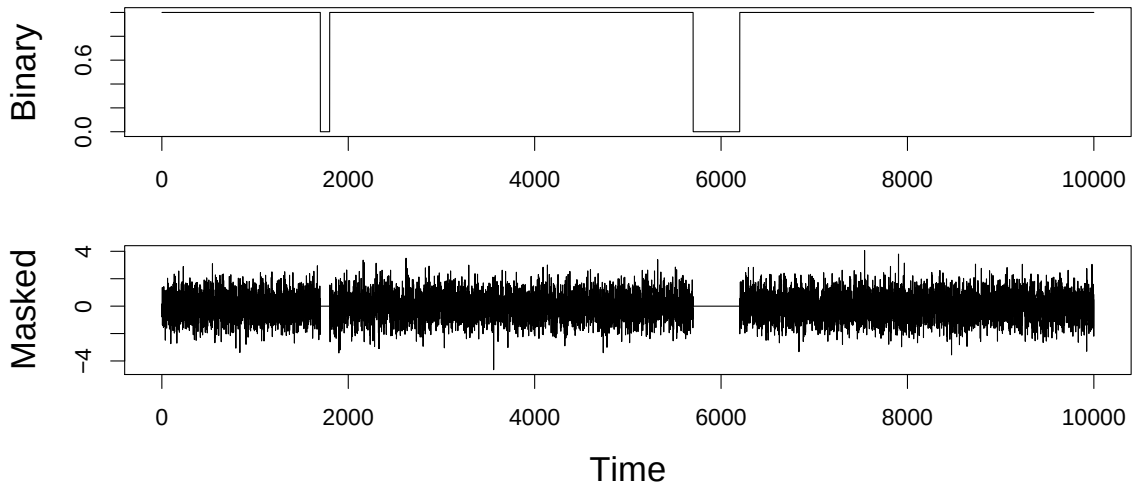



Figure 5.4: Applying the binary component (top) as a mask on `fan7` yields the masked component (bottom), whose masked values are replaced by the average of the unmasked values.

### 5.1.3 Query Syntax

A Qi query specifies what components to synthesize, the pairs of components for which to compute shared influence, and how to visualize the resulting relationships (via the optional parameters `arrow`, `edge`, and `top`). The syntax for specifying the focus, periphery, and synthetic components is a context-free grammar:

```
<query>   ::= ["periph="<regex>] <term> (" " <term>)*
<term>    ::= [<target> "<-"]<match>[<mask>]
<mask>    ::= "{" [!] <match> ("," <match>)* "}"
<target>  ::= <string> ("," <string>)*
<match>   ::= <regex> | <keyword> | <range>
<keyword> ::= "all" | "last" | <ctype>
<range>   ::= <time> ":" <time>
```

In this definition, `<regex>` represents a valid regular expression using standard syntax, `<time>` represents a numerical time value within the range spanned by the log, and `<ctype>` represents the name of some type of component within the system (e.g., `meta`, `alert`, or `normal`; see Section 5.1.2). The set of component types may be extended. An interval (e.g., `a:b`) can be used as a `<range>` (see Section 5.1.2) or

a `<mask>` (the optional `!` inverts the mask; see Section 5.1.2). The `all` keyword represents every component in the system and `last` stands for every component that appeared in the previously plotted SIG.

## 5.2 QI Implementation

We have implemented the query language in Section 5.1 as a tool called QI, written in Python.

QI spends the majority of the time computing cross-correlations and finding local extrema (see Section 5.1.1). There are algorithms for computing cross-correlation more quickly than the brute-force method: $O(n \log n)$ versus $O(n^2)$. These efficient algorithms compute the full cross-correlation function; in practice, however, delays above some maximum value are unlikely to be considered interesting. For example, anomalies in one component followed weeks later by anomalies in another component are unlikely to be considered related, regardless of the strength of the correlation. In such cases, where the ratio of the length of the anomaly signals to the maximum delay is sufficiently large, the brute force algorithm is actually faster. QI uses this ratio to decide which algorithm to use.

The cross-correlations are all mutually independent and can therefore be done efficiently in parallel (see Section 5.4.7). QI can be used offline by precomputing cross-correlations for so-called post-mortem analysis. The full correlation and delay matrices may be huge (billions of entries) but only have a subset of valid (already calculated) entries, so our implementation uses sparse matrices to store $\mathbf{C}$ and $\mathbf{D}$.

## 5.3 Systems

We evaluate QI using data from seven production systems: four supercomputers, one cluster, and two autonomous vehicles. Table 5.1 gives a summary of these systems and logs, described in Sections 5.3.1–5.3.4 and elsewhere [36, 69, 70]. For this wide variety of systems, we use QI queries to build influence models and to isolate a number of different problems. These systems were neither instrumented nor perturbed in any

| System | Components | Lines | Real 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 |

Table 5.1: The seven unmodified and unperturbed production systems used in our case studies. The 'Components' column indicates the number of logical components with instrumentation; some did not produce logs. Real time is given in days:hours:minutes:seconds.

way for our experiments.

### 5.3.1 Supercomputers

We use the four publicly-available supercomputer logs from Chapter 3. These four systems vary in size by several orders of magnitude, ranging from 512 processors in Liberty to 131,072 processors in BG/L. We make no modifications to the raw logs, whatsoever. An extensive study of these logs can be found in Chapter 2. 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
```

As before, we use the Nodeinfo algorithm (see Chapter 3) to generate anomaly signals from the raw textual 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 using the process described in Section 5.1.2. 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.

### 5.3.2 Mail-Routing Cluster

We obtained logs from 17 machines of a campus email routing server cluster mail-routing cluster at Stanford University. 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.

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.

### 5.3.3 Autonomous Vehicles

We use logs from the two autonomous vehicles introduced in Chapter 4. Recall that 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 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 the method from Chapter 4, which is based on anomalous distributions of message interarrival times. We generate no indicator or aggregate signals for the vehicles.

### 5.3.4 Log Contents

Table 5.2 provides some concrete example messages from different types of components in the systems we study. Logs include messages like the Spirit admin example, which indicates correct operation; the BG/L compute example, which indicates that a routine problem was successfully masked; the Thunderbird compute example, which indicates a real problem that requires administrator attention; and the BG/L admin example, which ambiguously suggests that something might be wrong. Some

| System | Component Type | Example Message |
|---|---|---|
| Blue Gene/L | compute | RAS KERNEL INFO total of 12 ddr error(s) detected and corrected |
| | admin | NULL DISCOVERY WARNING Node card is not fully functional |
| Thunderbird | compute | kernel: scsi0 (0:0): rejecting I/O to offline device |
| | admin | kernel: Losing some ticks... checking if CPU frequency changed. |
| Spirit | compute | kernel: 00 000 00 1 0 0 0 0 0 0 00 |
| | admin | sshd[11178]: Password authentication for user [username] accepted. |
| Liberty | compute | kernel: GM: LANAI[0]: PANIC: mcp/gm_parity.c:115:parity__int():firmware |
| | admin | src@ladmin2 apm: BIOS not found. |
| Mail Cluster | mail | postfix/smtpd[3423]: lost connection after DATA (0 bytes) from unknown[[IP]] |
| | pmx | [3999,milter] 4AB9C565_3999_1743011_1: discard: [IP]: 100% |
| Junior | sensor | RADAR1 25258 6 6 1 1 10.562500 [...] 0 0 51 1194100038.298347 kalman 0.166294 |
| Stanley | sensor | IMU -0.003300 -0.051810 [...] 0.109846 -0.030222 1128780826.436368 rr1 52.536104 |

Table 5.2: Example messages from different types of components in the systems we studied. There are no representative messages, but these are not outliers. Bracketed text indicates omitted information; the component names and message timestamps are removed.

messages, like the Liberty compute example, provide specific information about the location of a problem; others, like the Liberty admin example, state a symptom in (mostly) English; finally, some messages, like the Spirit compute example, appear incomprehensible without the (unavailable) source code.

The logs do not contain information about message paths (senders or recipients), function call or request paths, or other topological hints. These are production systems, so none were configured to perform aggressive (so-called 'DEBUG-level') logging

or to record detailed performance metrics (e.g., minute-to-minute resource utilization). Some messages are corrupted or truncated.

A system may have dozens of different types of components, and even individual components may generate hundreds of different types of messages. The content of these logs sometimes changes when software or hardware is updated or when workloads vary, and such changes may not be explicitly recorded in the log.

These logs exemplify the noisy and incomplete data available to system administrators working with real production systems.

## 5.4 Results

We present results in this section as a series of use cases. These examples both exhibit interesting features of QI and demonstrate that our method can isolate the sources of non-trivial bugs in a variety of real systems. Most of these queries take only a few seconds to execute; the runtime of each query is listed to the right of the query and collected at the end of the section in Table 5.3. (For each system, we start with no computations performed and execute exactly those queries in the order shown, on an 84-core cluster.) Section 5.4.7 explains why QI scales well under realistic usage.

QI outputs graphs in the DOT language that use layout features to communicate information about the SIG: edge thickness proportional to the strength of the correlation, node shapes and colors according to the type of component (e.g., binary component or metacomponent), grayed component names to indicate whether they are in the focus or periphery, and so on.

In this chapter, however, we convert these graphs to a manual layout for conciseness and readability, while retaining some of the visual cues. In our plots, edges touching grey rectangles denote a similar edge touching each contained vertex. We denote cliques (fully connected subgraphs) of four or more nodes as a small box: all nodes connected to it are in the clique. Disconnected components are omitted. Shaded nodes are in focus; unshaded nodes are in the periphery. Rectangular vertices represent synthetic components; vertices are elliptical, otherwise.

We discussed the following results with the administrators of the respective systems. Universally, the administrators felt the SIGs were interesting and useful; in some cases, the results were surprising and led the administrator to ask follow-up questions or take direct action (such as deciding to add or remove instrumentation). Administrators often have a mental model of how system components are interacting, which a given SIG will either reinforce or contradict. For example, one cluster administrator using the output of QI to debug a recurring but elusive database problem said the following:

> Yes, that [SIG] *is* intriguing. In general, I really liked this graph... [because] it provides an interesting picture of the related components of a system... The link between slow queries and threads, established connections, and open files used confirms for me a suspicion that the root of MySQL performance problems for us are slow queries, and that we get spikes in utilization when we have slow queries. That's useful information... [The SIGs seem to] rule out, or at least make less likely, the theory that a sudden surge in www activity was what set off the MySQL problem. That was one of our working theories, so knowing that's less likely is valuable.

There were no instances of QI inferring a shared influence where there certainly was none (no false positives), nor any known instances of QI failing to detect shared influence where there certainly was some (no known false negatives).

### 5.4.1 Alert Discovery on Blue Gene/L

When a system is too large to consider components individually, one can use metacomponents: synthetic components that represent the aggregate behavior of sets of components (see Section 5.1.2). A metacomponent is specified by the set of components it represents, which may be in the form of regular expressions (i.e., all components matching that regex). On Blue Gene/L (BG/L), we defined one metacomponent for each rack of the system; the components are named according to their topological


Figure 5.5: The anomaly signal of the synthetic `CRASH` component, which encodes when a job on BG/L crashed.

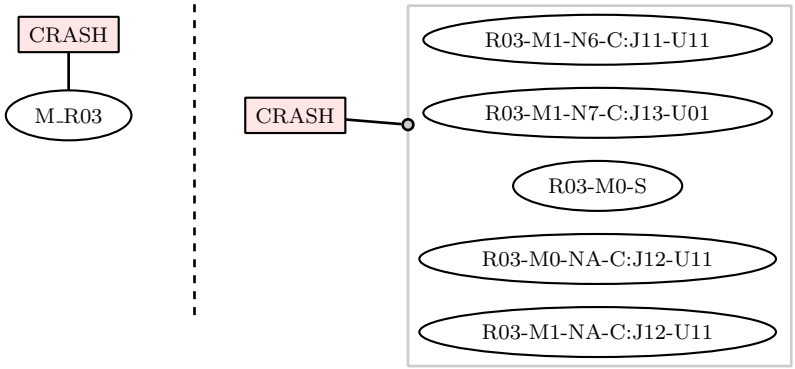



Figure 5.6: Components most strongly correlated with the crash. On the left, the metacomponent; on the right, the components within that metacomponent.

location, so rack 47, for example, can be made into metacomponent `M_R47` using the regex `R47`.

Consider the (real) scenario when a full-system job running on BG/L crashes and the administrator knows approximately when. Initially, every component of the system is a possible cause of the failure. Using QI in conjunction with metacomponents, we show how to iteratively isolate the components that are likely to be involved. The administrator can execute the following query, which constructs a binary component using the time of the crash as the focus and using the metacomponents, collectively, as the periphery. This is asking, at a coarse granularity, what large subsystem's aggregate anomaly signal is most strongly correlated with the observed crashing behavior:

```
top=1 periph=meta CRASH<-174:175
```
(1.65 sec)

This query creates a synthetic component from the interval beginning 174 hours

into the log and ending one hour later and names it `CRASH`. Figure 5.5 shows the anomaly signal for this synthetic component. The result of the query is on the left in Figure 5.6. Qi implicates rack 3 (`M_R03`), so we then ask for a short list of the components in rack 3 that share an influence with the crash:

`top=5 periph=R03 CRASH` (4.81 sec)

Recall that Qi does not compute the relationships between pairs of components in the periphery. The result, plotted on the right in Figure 5.6, shows the five components with the strongest correlation, plotted from top to bottom in order of decreasing strength. In other words, node 6, in midplane 1 of rack 3, seems to be most strongly related with the problem; its neighbor, node 7, also seems suspicious. (Running the first query with `periph=normal`, i.e., all non-synthetic components, instead of `periph=meta`, leads to the same conclusion but takes more than 1000 times as long.) Once Qi has implicated a particular component around a particular time, the user can simply inspect the corresponding section of the log; in the case of BG/L's textual logs this was a simple `grep` for the component name and time. During the time surrounding the crash, 90 other components generated hundreds of messages, but the suspect node only generated two. The following one of those messages is considered an "actionable alert," meaning it is a problem the administrator wants to know about and can do something to fix: `ddr: Unable to steer [...] consider replacing the card`. None of the other messages generated by any of the other components were alerts: node 6 was likely the responsible component.

Once the administrator is aware of specific messages, such as this DDR error, they can look for them, explicitly. Discovering these messages in the first place, however, is often a key part of the administrator's job. This BG/L example shows how Qi can facilitate that discovery process.

### 5.4.2 Correlated Alerts on Liberty

Given an alert message, such as the one in Section 5.4.1, we can make synthetic components with anomaly signals that represent the presence or absence of that type of message. Specifically, given a regex that identifies whether or not a message is an

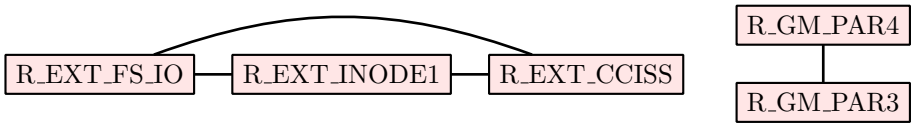



Figure 5.7: Some alert types on Liberty are correlated; QI helps search for the reasons why.

instance of each type of alert, QI can automatically generate a synthetic component indicating whether the alert was generated anywhere in the system (identified by the name of the alert type) or by a particular component (identified by the alert type concatenated by a forward slash onto the component name). So, a synthetic component named, by convention, `node5/ERR` has an anomaly signal that is high exclusively when component `node5` generated an alert of type `ERR`; telling QI to generate such a component is simply a matter of writing a regular expression that describes `ERR` alerts.

Using the alerts identified for the Liberty data set in Section 2.1.2, we show how QI can elucidate relationships among these synthetic alert components (identified by the keyword `alert`). The command

`edge=0.1 alert` (1.12 sec)

generates Figure 5.7, revealing the relationships among alert types. Most alert types are not correlated with each other (thus, omitted from the graph); however, there are also clusters of related alerts.

We might then ask, say, whether the clique of alerts containing the `EXT` string (on the left in Figure 5.7) are truly redundant. To look at when individual components generated these `EXT` strings, we can use the string as a regular expression:

`periph=EXT last` (1.00 sec)

This generates Figure 5.8, which shows that the `CCISS` alert is sometimes seen without the other two `EXT` alerts, meaning it is likely generated under a wider variety of conditions, and that `node176` tends to generate `CCISS` alerts at times when it is also generating `GM` alerts. Both are checksum- or parity-related errors, so a common source of data corruption would be a good place to continue the search.

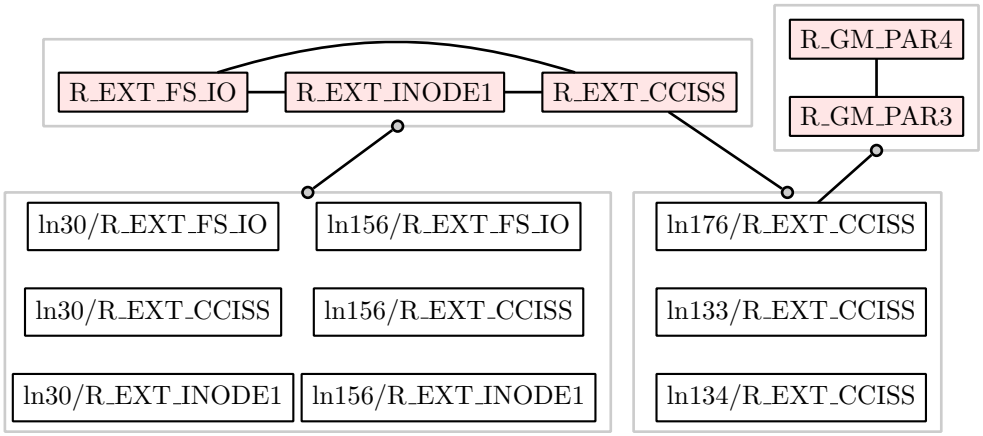



Figure 5.8: Synthetic components can show the relationships among both when certain alert types are generated and when they are generated on individual components.

### 5.4.3 Obscured Influences on Spirit

Large systems often experience multiple simultaneous problems. For instance, a supercomputer node may generate strange messages both because of a disk malfunction and because of an unrelated software glitch. Using masked components, QI can elucidate which shared influences result from which problem. For example, if we mask the portions of the anomaly signal that correspond with disk errors, we may see more clearly what shared influences result from the software bug.

Say that we want to investigate the behavior of node `sn138` on the Spirit supercomputer, which generated both disk errors and batch scheduler errors. We might first execute the following:

```
edge=0.25 top=5 periph=normal sn138
```
(2.19 sec)

The graph in the top left of Figure 5.9 shows the result: there is exactly one other node that seems correlated with `sn138`. Based on our study of this log (see Chapter 2), we are aware of many of the kinds of alerts that occur on this system. One, called `PBS_CHK`, is a batch scheduler alert; the other, `EXT_CCISS`, is a file system alert.

Do either of these alerts account for the shared influence between `sn138` and `sn487`? One way to pose this question to QI is by masking the contribution of one alert and repeating the question:

```
periph=normal sn138{!sn138/PBS_CHK}
```
(2.05 sec)

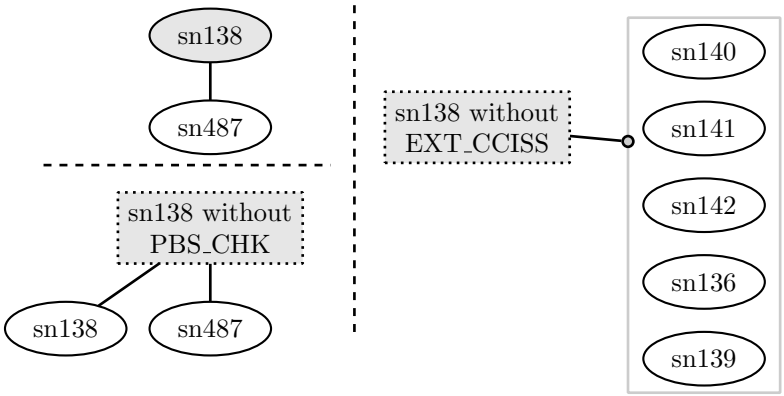



Figure 5.9: Masking the contribution of one anomaly source can make other shared influences more apparent.

The other parameters inherit their values from the previous query. Because the mask portion of the query begins with the '!' character, it means we retain the sections of `sn138`'s anomaly signal only where that node did not generate the `PBS_CHK` alert. The result is shown in the bottom left of Figure 5.9. As expected, the masked version of `sn138` correlates perfectly with the original; the masked portions of the anomaly signal do not contribute to the cross-correlation while the rest matches perfectly. Meanwhile, the shared influence with `sn487` remains strong: the `PBS_CHK` alert is not driving the correlation.

We ask the analogous question about the disk errors:

`periph=normal sn138{!sn138/EXT_CCISS}` (2.03 sec)

As seen in the graph on the right in Figure 5.9, a new set of nodes exhibits a shared influence with `sn138`; node `sn487` doesn't make the top-5. This means both that the disk errors account for some of the shared influence between `sn138` and `sn487` and that these errors were obscuring additional shared influences—revealed by the query—between `sn138` and other components.

### 5.4.4 Thunderbird's "CPU" Bug

When a problem is systemic or involves multiple components, the text of log messages may be misleading because they reflect only locally observable state. Nevertheless,

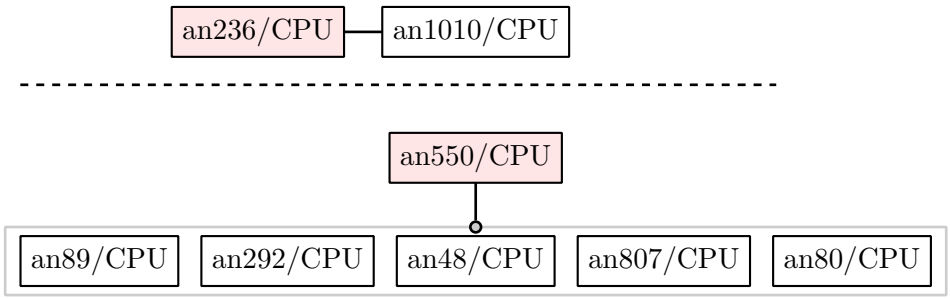



Figure 5.10: Binary components representing the 'CPU' alert tend to share a strong influence with sets of such components that are topologically close, such as those in the same job scheduling group.

these superficially misleading messages may still be useful for understanding a problem, as we demonstrate with the following example from the Thunderbird supercomputer.

Thunderbird occasionally generated the following alert message:

```
kernel: Losing some ticks... checking if CPU frequency changed.
```

Although ostensibly a processor-related issue, the underlying cause of the message was actually a bug in the Linux SMP kernel that would cause the OS to miss interrupts during heavy network activity. The key insight for isolating this bug was that these "CPU messages" were spatially correlated; when one node generated the message, it was more likely that other, topologically nearby nodes would also generate it. These groups of nodes corresponded to job scheduling groups, which implicated particular workloads as a possible trigger of the alert.

We now show how, using the alert message as an initial clue, Qi helps elucidate these spatial correlations. First, construct binary components for each component that generated the 'CPU' alert and one for the 'CPU' alert for the whole system, as described in Section 5.4.2. Second, take one of these binary components—say, `an236/CPU`, the synthetic component for node `an236` representing when it generated the 'CPU' alert—and compute how it relates to the other binary components:

```
edge=0.25 top=5 periph=CPU an236/CPU                    (2.93 sec)
```

As shown on the top in Figure 5.10, `an236/CPU` shares a strong influence with

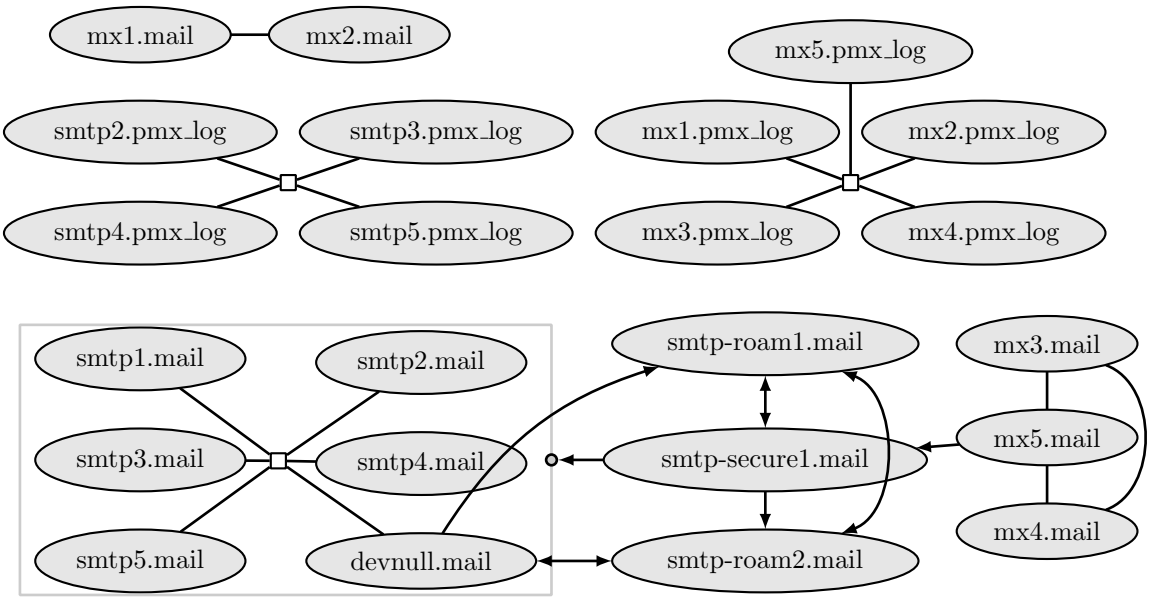



Figure 5.11: Shared influence in a cluster of mail-routing servers.

`an1010/CPU` but not with the others. To convince ourselves that this is not a coincidence, we take another component and repeat:

`periph=CPU an550/CPU` (2.66 sec)

This yields the SIG on the bottom in Figure 5.10, where `an550/CPU` shows shared influences with other binary components. Notably, however, these correlated alerts often seem to occur on the same rack. Indeed, if we examine large groups of these 'CPU' binary components, we learn that they tend to form cliques that are related to their topology; as explained above, this is sufficient to rule out a local CPU malfunction and to suggest a common cause.

### 5.4.5 Mail-Routing Cluster

Even in the absence of some known bad behavior, Qi can be used to model the flow of influence in the system. We examine the influence among all components in the mail-routing cluster described in Section 5.3.2 by executing the following:

`arrow=1 top=47 all` (2.36 sec)

This command considers all components in the system, plots the strongest 47 edges (this choice is not significant; we picked what fit in the figure), and assigns directionality to any influence with a delay of one minute or more. The result, shown in Figure 5.11, gives an overview of the influences in this cluster. Note that we determined these shared influences and temporal orderings even without knowledge of system topology, message paths, or request sequences.

When we showed this graph to the cluster administrator, his initial response included the following:

> Similarly, having all the smtp mail servers linked makes sense. But I'm surprised that devnull is linked in with them. I assume that must be due to mail from one Stanford user going to the vacation or autoresponder system on devnull, but I'm surprised that the same relationship isn't there for the mx servers. I think that may say something interesting about where most of the hits on the vacation and autoresponder services come from.

Such questions and suspicions led to follow-up queries, the results of which he described as illuminating and valuable.

### 5.4.6 Stanley's Swerve Bug

Temporal ordering is sometimes crucial to understanding a problem, as we demonstrate with an example from an autonomous vehicle. On several occasions during the Grand Challenge race, Stanley appeared to swerve around a non-existent obstacle. This bug, described in Chapter 4 and elsewhere [70], was caused by a buffer component shared by the laser sensors, which passed stale data to the downstream software. Although this shared component was not instrumented to generate log messages, Chapter 4 explained how to use SIGs in an ad hoc way to isolate a shared component of the lasers as a likely cause.

We now show how to use Qi to isolate the bug more systematically, given only the clue that most manifestations of the `SWERVE` bug occurred between mile-markers 22 (around 60 minutes in) and 35 (around 100 minutes in). In Qi syntax, we could generate the plot from Chapter 4 (including the same edge and arrow thresholds),

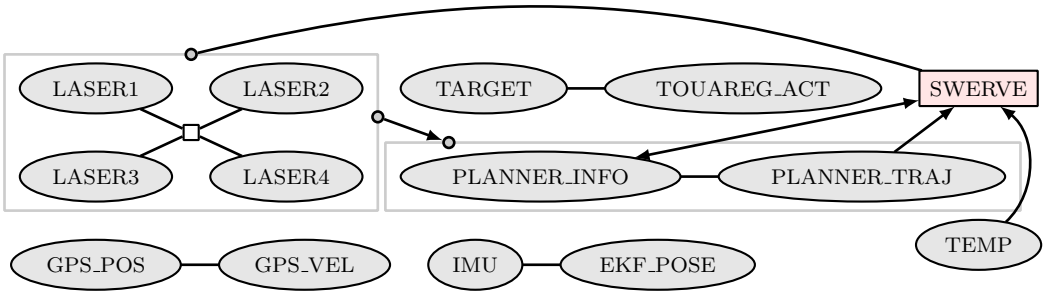



Figure 5.12: A complete SIG for Stanley, including a synthetic component, `SWERVE`, that represents an unexpected behavior.

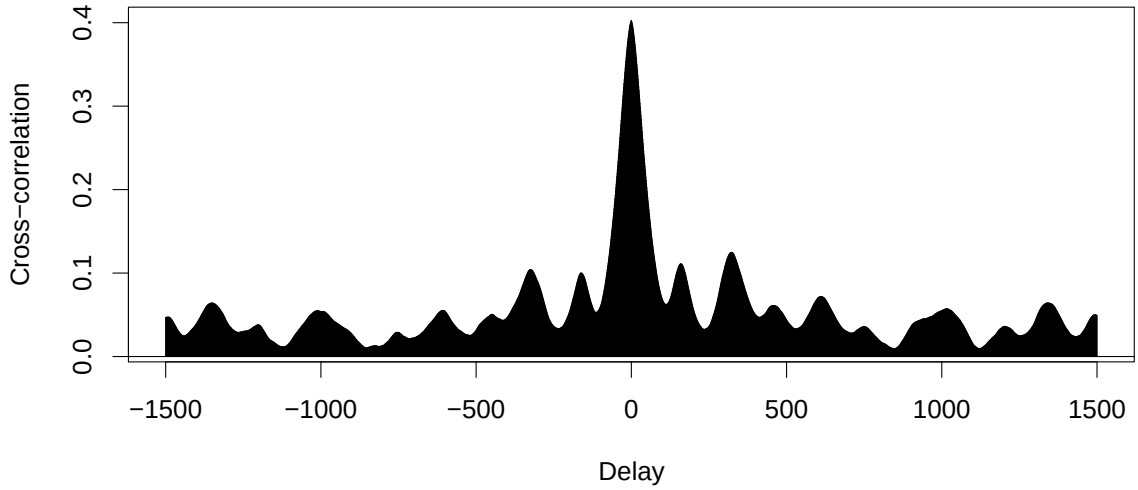



Figure 5.13: Cross-correlation between `LASER1` and `LASER2`.

shown in Figure 5.12, using the following command:
`arrow=90 edge=0.15 all SWERVE<-60:100`.

Figure 5.13 shows the cross-correlation of `LASER1` with `LASER2`: a strong correlation at zero delay, resulting in an undirected edge in the SIG. Using the data from Stanley's successor vehicle, Junior, we were able to verify that the lasers no longer shared an influence—the buffer component was no longer shared among all the lasers.

Although Figure 5.12 contains the information that helped isolate the bug on Stanley, it also contains irrelevant information: components not on any influence pathway with `SWERVE`. QI can generate graphs that omit such noise. What the user

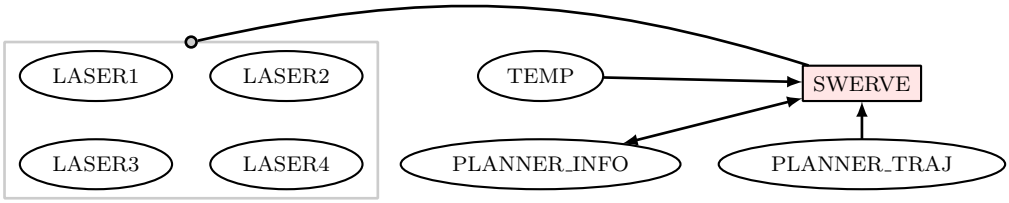



Figure 5.14: The components most strongly correlated with the swerving behavior.

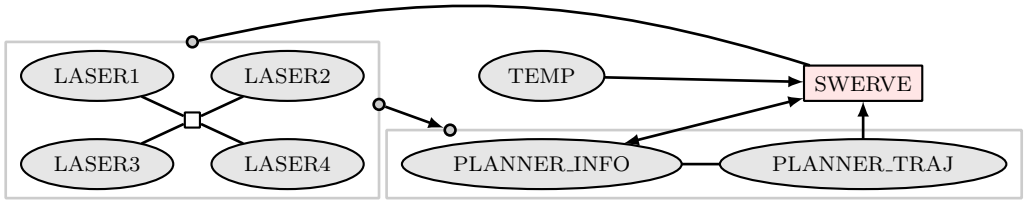



Figure 5.15: This automatically generated graph implies that a shared component of the lasers is likely to be causally related to the swerving.

really intends to ask is, "What components seem to be most strongly related to this swerving behavior?" This is expressed in QI as the following query, which generates the SIG shown in Figure 5.14:

```
periph=all SWERVE
```
(20.37 sec)

QI queries for Stanley and Junior tend to take longer than the other systems because their anomaly signals have a finer time granularity.

A natural follow-up question, given the components that seem to share a strong influence with the swerving behavior, is what influences are shared among *those* components. This is a simple query that instructs QI to compute a graph in which all the components from the previous SIG are in focus:

```
last
```
(35.60 sec)

The result, shown in Figure 5.15, contains exclusively the components and interactions relevant to the swerving bug, with the exception of `TEMP` (see below). The clique of laser sensors implies an (uninstrumented) shared component, and the directionality of the arrows from the lasers to the planners and then the planners to the swerving implies causality. The Stanford Racing Team tells us that QI would likely have saved them two months of debugging time for this problem, alone.

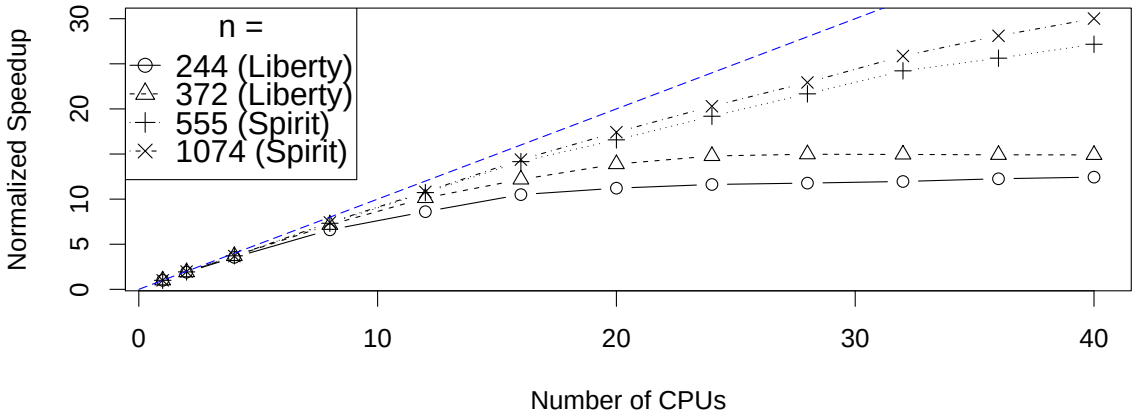



Figure 5.16: Speedups for components on Liberty and Spirit, normalized according to the runtime on a single local CPU. Those baselines were 117.31 sec, 276.18 sec, 1208.64 sec, and 4611.76 sec, respectively. The legend indicates the number of signals ($n$), which we varied to show the effect of system size.

The temperature sensor is actually anti-correlated with `SWERVE`. A more precise definition of `SWERVE` that included all swerving incidents would have eliminated this spurious correlation, which occurs because the `SWERVE` anomaly signal is non-zero only near the beginning of the race while the `TEMP` anomaly signal increases over the course of the race (`TEMP` corresponds to a temperature sensor, and the desert day grew hotter as the race went on).

### 5.4.7 Performance and Scaling

We have used QI with systems containing as many as 69,087 log-generating components (see Section 5.4.1). It is impractical to compute all pair-wise correlations among these components; one important contribution of QI is the ability to ask queries that will only compute a relevant subset of those pairs. Furthermore, each pair (i.e., each cross-correlation) can be computed independently of every other; the task is embarrassingly parallel. QI exploits this parallelism to achieve nearly linear speed-ups (see Figure 5.16). Even for queries with more than a thousand components in focus, if we have access to forty cores then QI can complete the query in a couple of minutes. For realistic queries, like those of Sections 5.4.1–5.4.6, the summary of runtimes in

| System | Command | Time (sec) | #CCs |
|---|---|---|---|
| **Blue Gene/L** | `periph=meta CRASH<-174:175` | 1.65 | 64 |
| | `periph=R03 CRASH` | 4.81 | 1084 |
| **Liberty** | `alert` | 1.12 | 55 |
| | `periph=EXT last` | 1.00 | 54 |
| **Spirit** | `periph=normal sn138` | 2.19 | 520 |
| | `periph=normal sn138{!sn138/PBS_CHK}` | 2.05 | 521 |
| | `periph=normal sn138{!sn138/EXT_CCISS}` | 2.03 | 521 |
| **Thunderbird** | `periph=CPU an236/CPU` | 2.93 | 1031 |
| | `periph=CPU an550/CPU` | 2.66 | 1030 |
| **Mail Cluster** | `all` | 2.36 | 528 |
| **Stanley** | `periph=all SWERVE<-60:100` | 20.37 | 16 |
| | `last` | 35.60 | 120 |

Table 5.3: The time taken to execute the commands shown on a cluster of 84 cores. The '#CCs' column indicates the number of cross-correlations computed.

Table 5.3 shows that QI can be used interactively.

## 5.5 Contributions

We have presented a query language and implementation (QI) for understanding component behaviors and interactions in large, complex systems where instrumentation may be noisy and incomplete. Unlike previous work, QI requires no modifications to existing instrumentation and does not assume fine-grained or precise measurements (such as message paths). We found the capability to encode external knowledge using metacomponents and binary components to be especially useful. Although masked components serve a similar role, they lack the generality to express the variety of ways in which we might want to "ignore" certain behaviors. The ability to parallelize the queries was invaluable.

Using raw data from seven unmodified production systems, we demonstrated the use of QI for such tasks as alert discovery (see Section 5.4.1), problem isolation (see Sections 5.4.3, 5.4.4, and 5.4.6), and general system and interaction modeling (see Sections 5.4.2 and 5.4.5). As we demonstrated, our method scales linearly with system size and is fast enough to be used interactively for typical use cases (see Section 5.4.7).

As systems trend toward more components and more sparse instrumentation, methods like ours—with only weak requirements on measurement data and good scaling properties—will become increasingly necessary for understanding system behavior.

# Chapter 6

# Online Algorithm

This chapter presents a real-time method for computing influence that builds on the results of Chapters 4 and 5. Answers to some of the most important reliability 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. Furthermore, online algorithms inherently require fewer resources and therefore tend to be more readily deployed in a production context.

Our method uses 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 the approach are both new.

As before, a system is represented by a set of real-valued functions of time called *anomaly signals*, which encode when measurements differ from typical or expected behavior. 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) [48]; 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

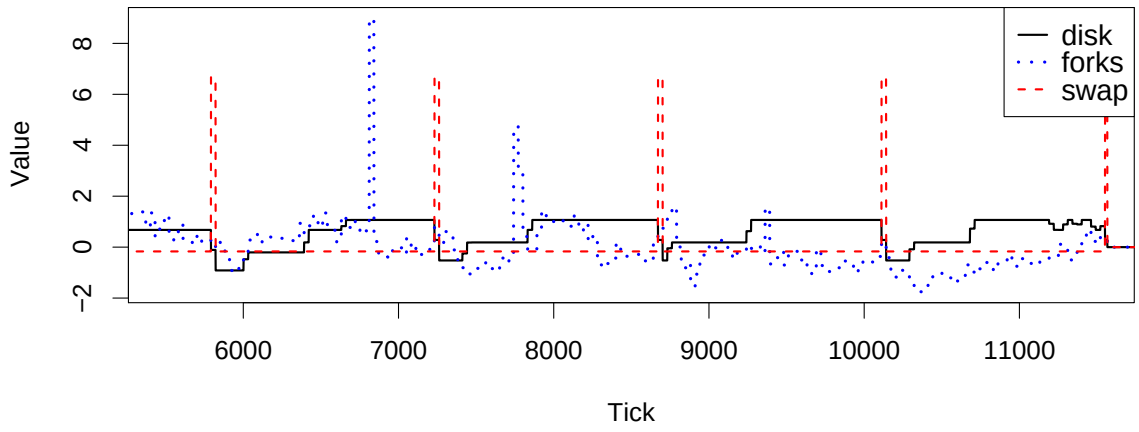



Figure 6.1: Three example anomaly signals. Greater distance from zero (average) corresponds to more surprising measurements. The signals `disk` and `forks` are statistically correlated.

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 6.1.3), and looks for lag correlations among them using an online approximation algorithm [56]. Although the eigensignals are mutually uncorrelated by construction, they may be correlated with some lag.

Figure 6.1 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 6.2, 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

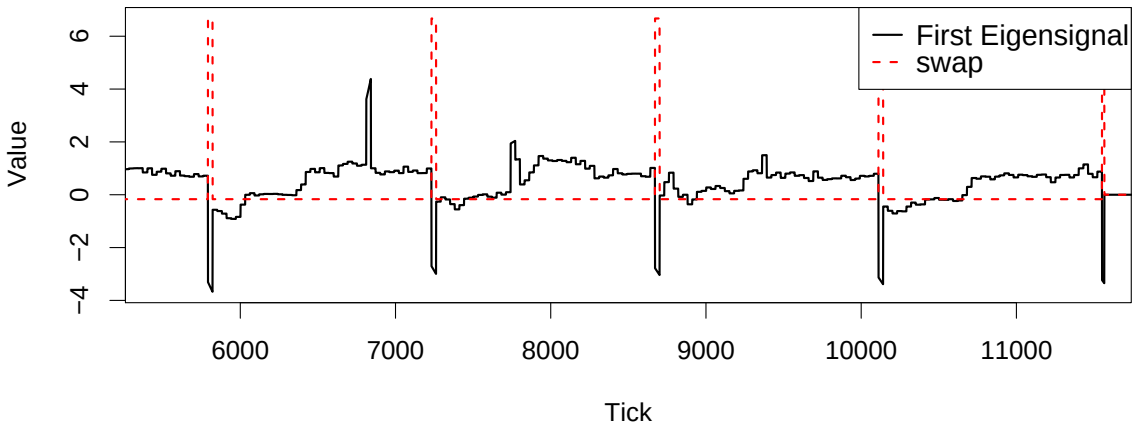



Figure 6.2: The first eigensignal and `swap`. The downward spike in the eigensignal consistently happens just before the spike in `swap`.

impending collapse for the administrator to take remedial action (see Section 6.3.4).

We describe our method in Section 6.1 and evaluate it using nearly 100,000 signals from eight unmodified production systems (described in Section 6.2), including four supercomputers, two autonomous vehicles, and two data center clusters. Our results, in Section 6.3, 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.

## 6.1 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

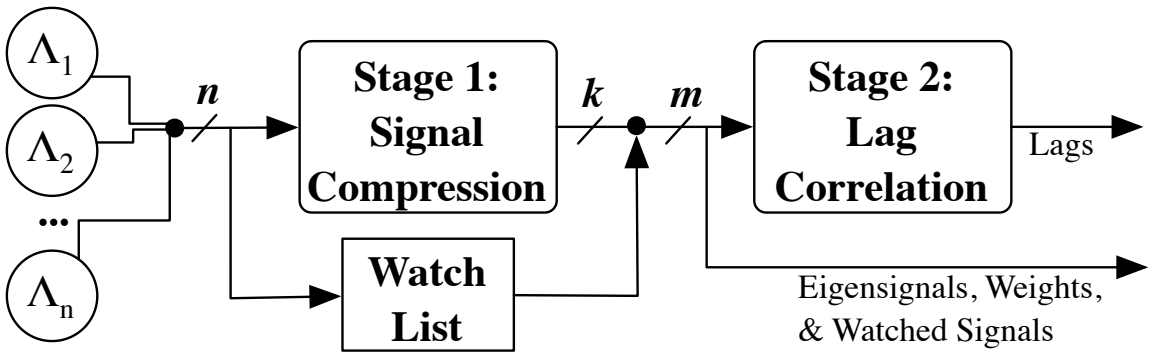



Figure 6.3: Our method takes $n$ anomaly signals and passes them to a signal compression stage (performed using principal component analysis). It then takes the resulting $k$ eigensignals and those on the watch list and passes these $m$ signals to a lag correlation detector. The listed output values are all available at any time.

natural questions about a system in terms of time varying signals.

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 6.1.1). 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 6.1.2) 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 6.1.3). Figure 6.3 provides an overview of our approach.

Our method uses an online principal component analysis for signal compression that is adapted from SPIRIT [48] and a lag correlation detection algorithm called Enhanced BRAID [56]. We selected these techniques because they make only weak assumptions about the input data and have good performance and scalability characteristics. This chapter 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.

### 6.1.1 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.

Numerical measurements can be used as anomaly signals, directly, while some measurements require a processing step to make them numerical (see Derived Signals, below). 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 Indicator Signals, below).

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.

### 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 [76, 77]. We use the Nodeinfo algorithm (Chapter 3) for textual logs and an information-theoretic timing-based model (Chapter 4) 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 network 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 like system logs, 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.

**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 (see

Section 5.1.2). Although this is the simplest way to encode expert knowledge about a log, indicator signals have proven to be both flexible and powerful. Section 6.3.3 gives an example of how indicator signals can elucidate system-wide patterns.

**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 (see Section 5.1.2), and we see in Section 6.3.3 that these aggregate signals often summarize important behaviors.

### 6.1.2 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). 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 [48], 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 6. 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 because it has been our experience that the signals with significant weight in a subsystem are all well-correlated. This observation also justifies picking the most heavily weighted signal in a subsystem as the *representative* of that subsystem (see Section 6.3.3).

It is worth noting that Xu et al. recently also used principal component analysis in their work [77]; they use it to identify anomalous event patterns rather than finding related groups of real-valued signals.

### 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 [48]. 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 and for studying those changes over time (see Section 6.3.3); 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.

### 6.1.3 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*; 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 [56]. 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 6.3.4). 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 Watch List, below), lag correlation can often be computed more quickly than the PCA (see Section 6.3.1). 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 6. 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.

#### 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 specific signals and to associate results with specific components. When no components are on the watch list, the results are presented exclusively in terms of the subsystems. 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 6.3.3). 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.

### 6.1.4 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.

## 6.2 Systems

We evaluate our method on data from eight production systems: the seven systems described in Section 5.3 plus a 9-node SQL database cluster located at Stanford University. 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
```

| 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 6.1: The eight 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.

```
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 5.3.1. We aggregate disk-related logs in the SQL cluster into a signal called `disk`, memory-related logs into `memory`, etc.

For convenience, the eight systems are summarized in Table 6.1 and the anomaly signals are summarized in Table 6.2. It has been our experience that the results of our method are not strongly sensitive to choices of anomaly signals; for any reasonable choice of signals, our method tends to group similar components and detect similar lags.

## 6.3 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.

| 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 6.2: 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.

In this chapter, 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 [48]). 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 6.1.2 and 6.1.3, 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 6.3.1 describes the performance of our analysis in terms of time and Section 6.3.2 discusses the quality of the results; we focus in these subsections on the mechanisms of the analysis, rather than their applications. Then, in Sections 6.3.3–6.3.4, 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., the SIGs from Chapter 4); this section focuses instead on the information our method produces and the uses of that information.

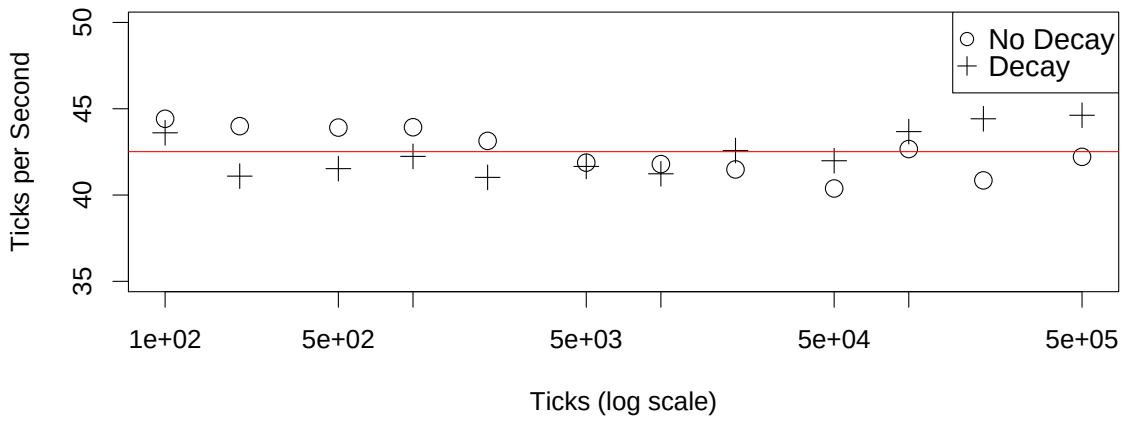



Figure 6.4: Using prefixes of Stanley's data ($n = 16$), we see that compression rate is not a function of the number of ticks.

### 6.3.1 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 6.4 and 6.5 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 6.6). 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 6.7). 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 processing rate to data

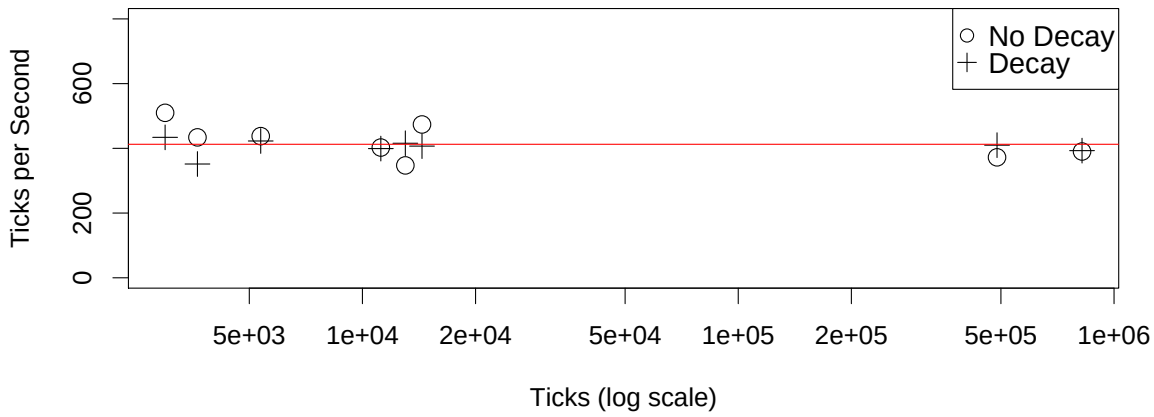



Figure 6.5: The lag correlation computation is not a function of the number of ticks ($n = 20$). Each pair of data points corresponds to one of our studied systems.

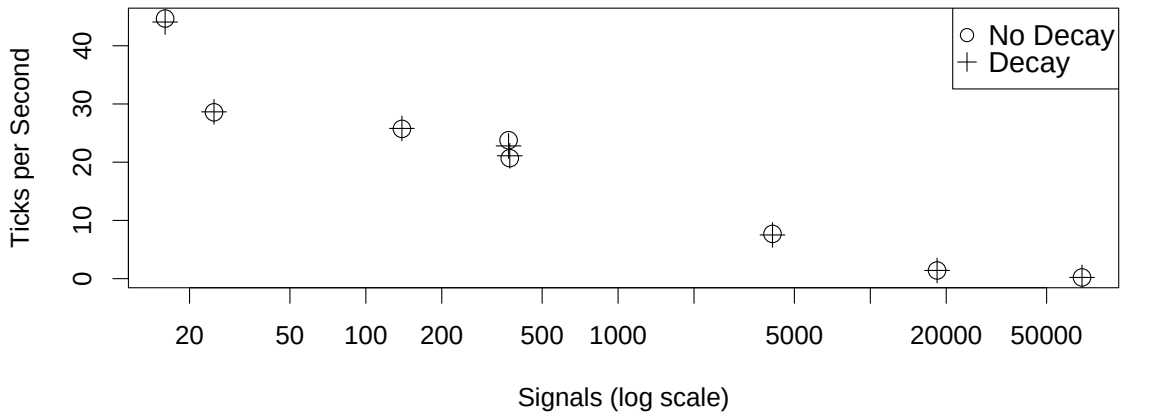



Figure 6.6: The rate of ticks per second for the compression stage decreases slowly with the number of signals; autoregressive weighting (decay) has no effect on running time.

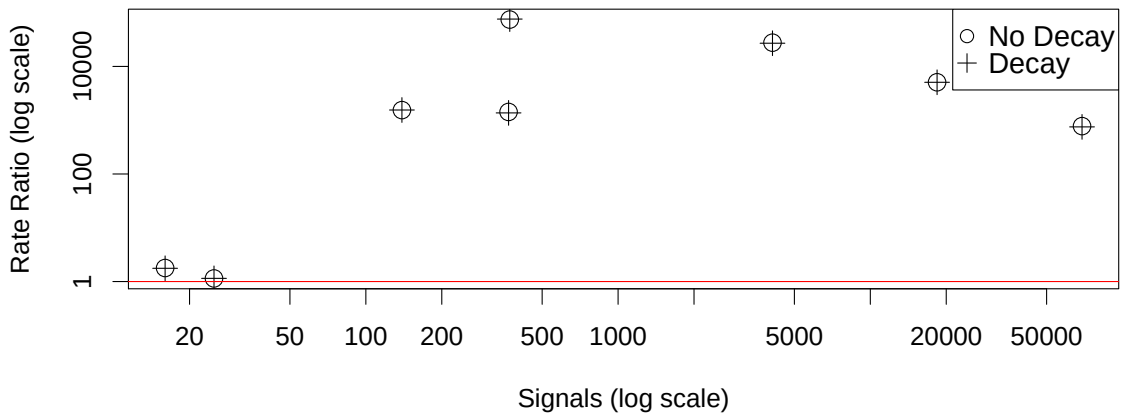



Figure 6.7: Although the compression rate decreases with the number of signals, larger systems tend to update measurements less frequently. The ratio between compression rate and measurement generation rate, plotted, shows that the bigger systems are easier to handle than the 25 ticks-per-second data rate of the embedded systems.

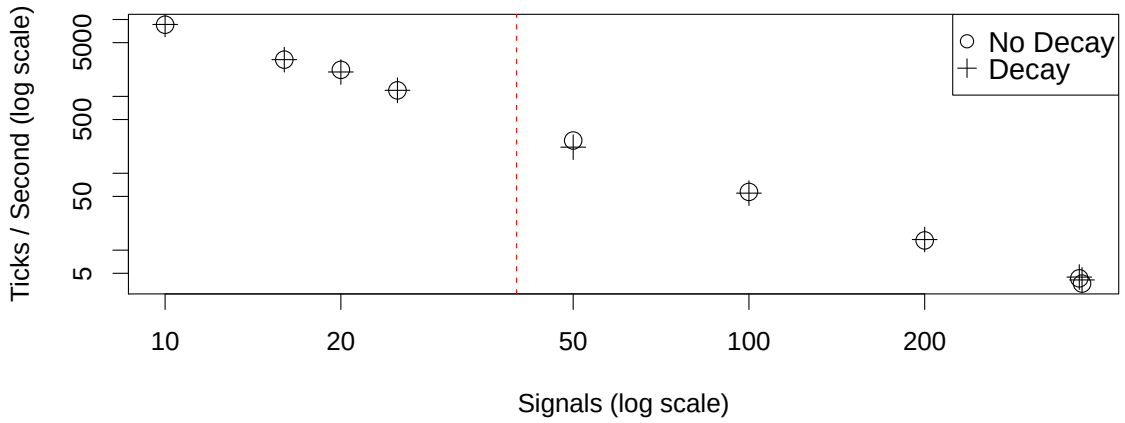



Figure 6.8: The rate of lag correlation processing decreases quickly with the number of signals. (Note the log-log scale.) Our method uses eigensignals and a watch list to keep the number of signals small.

generation rate, at 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 ($k$), 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 6.8 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 $m << n$.

### 6.3.2 Eigensignal Quality

Previous work uses a measure called *energy* to quantify how well the eigensignals describe the original signals [22, 48]. Let $x_{\tau,i}$ be the value of signal $i$ at time $\tau$. 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 6.9 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

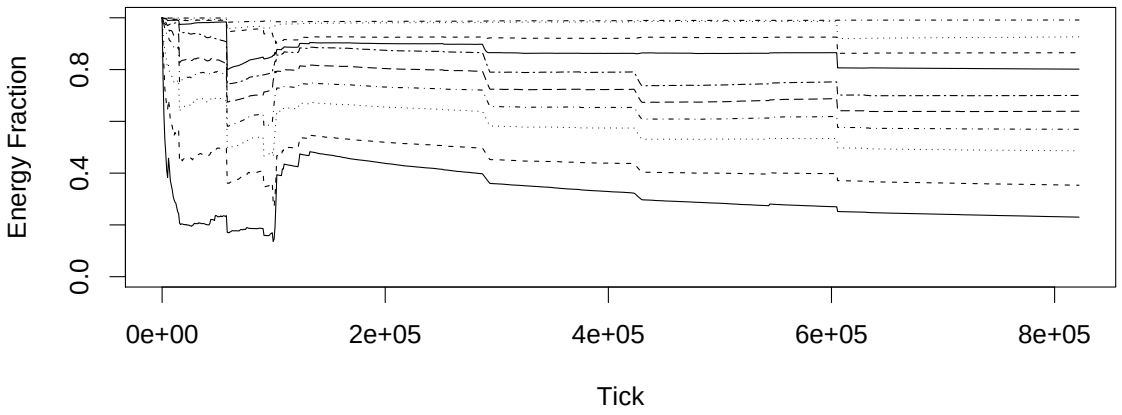



Figure 6.9: The cumulative fraction of total energy in Stanley's first $k$ eigensignals. The bottom line shows the energy captured by the first eigensignal; the line above that is for the first two eigensignals, etc.

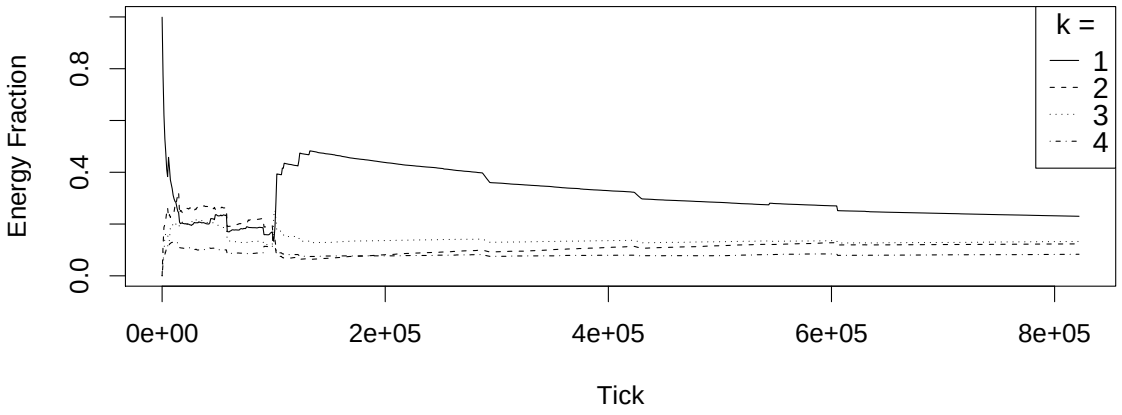



Figure 6.10: The incremental additional energy captured by Stanley's $k^{th}$ eigensignal, given the first $k-1$.

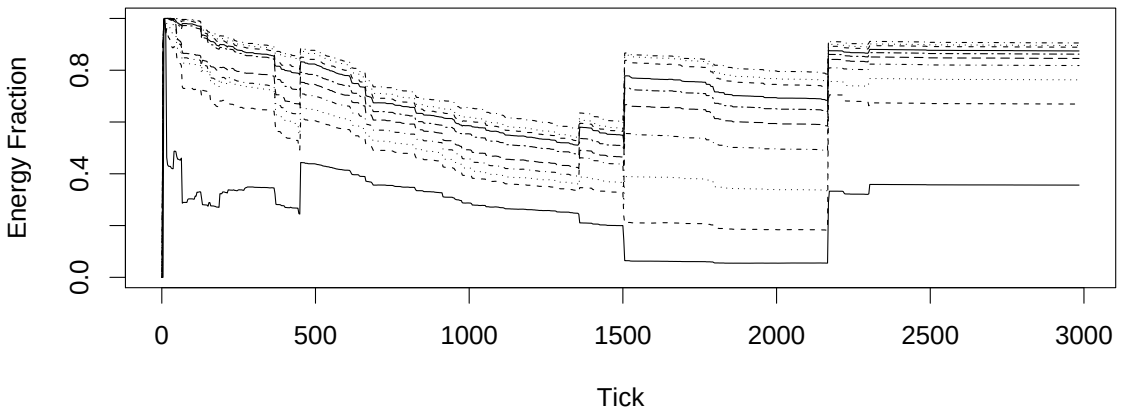



Figure 6.11: The cumulative fraction of total energy in BG/L's first $k$ eigensignals. The first ten eigensignals suffice to describe more than 90% of the energy in the system's 69,087 signals.

first three, and so on. Figure 6.10 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 6.11, which shows that the first eigensignal, alone, contains roughly 33% of all of the energy in the system.

Figure 6.12 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 6.13), twenty eigensignals is enough to bring the

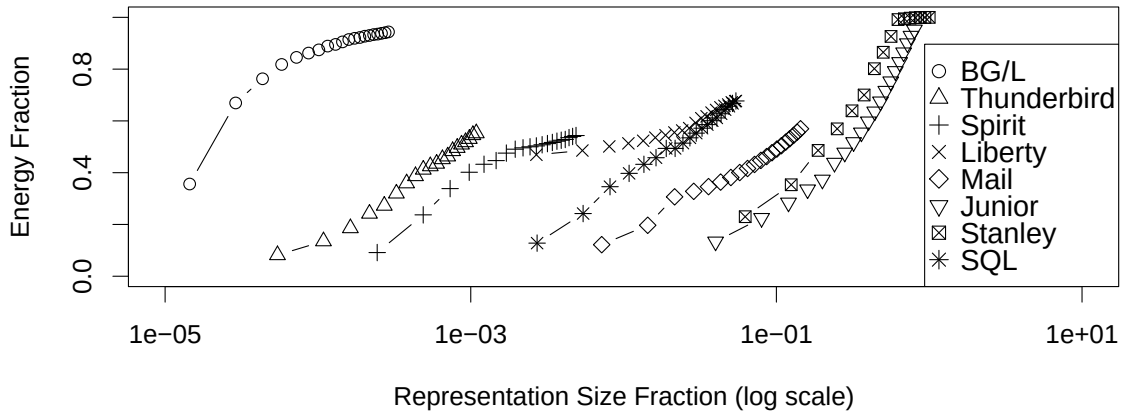



Figure 6.12: The fraction of energy captured by the first 20 eigensignals, plotted versus the size of those signals as a fraction of the total input data. (Note that Stanley only has 16 components and therefore only 16 eigensignals.)

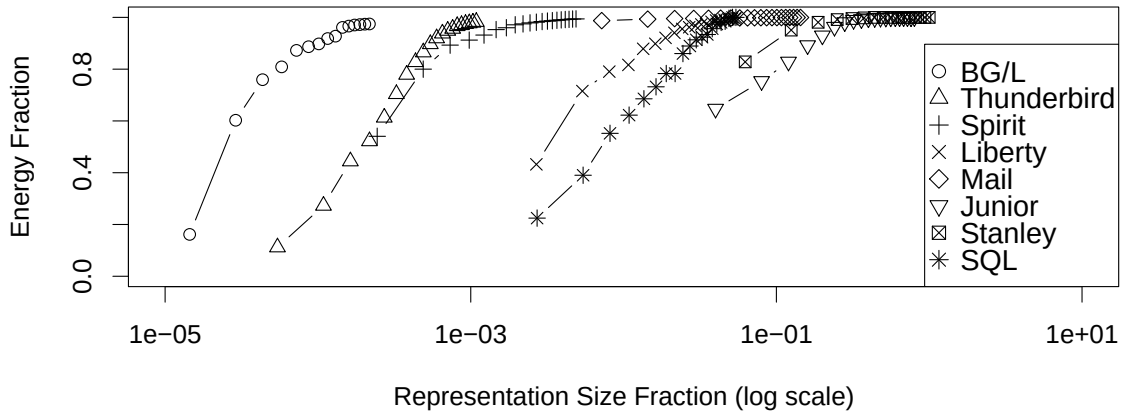



Figure 6.13: When old data is allowed to be forgotten (decay), the behavior of the system can be described efficiently using a small number of eigensignals.

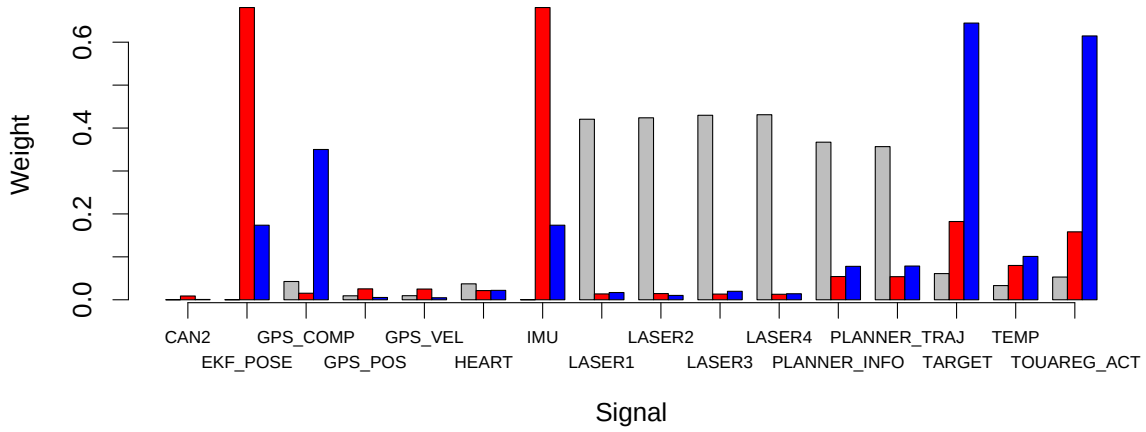



Figure 6.14: Weights for Stanley's first three subsystems. The left bar indicates the absolute weight of that signal's contribution to the subsystem; the second bar indicates its weight in the second subsystem, etc.

energy fraction to nearly one; for the larger systems, this means we are compressing by orders of magnitude with minimal information loss.

### 6.3.3 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.

#### Identifying Subsystems

During the Grand Challenge race, Stanley experienced a critical bug that caused the vehicle to swerve around nonexistent obstacles [70]. 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 6.14). 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

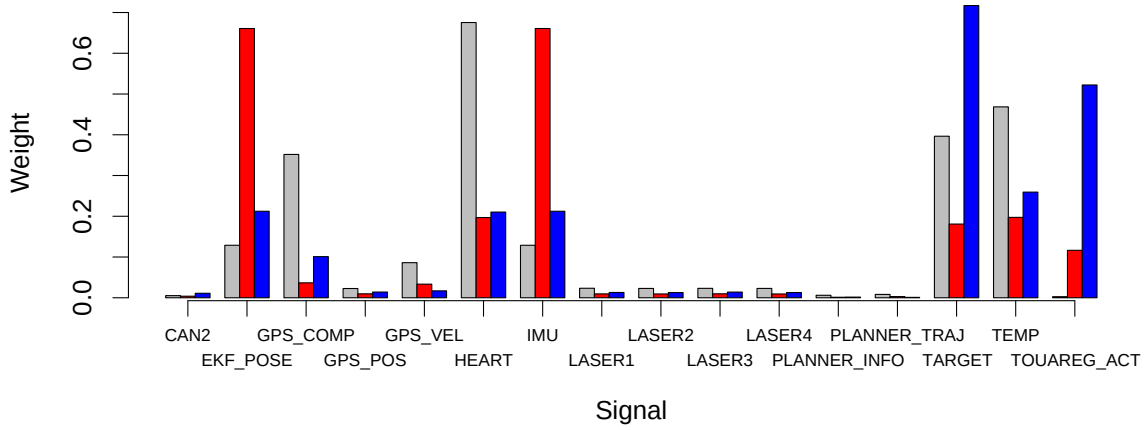



Figure 6.15: Weights of Stanley's first three subsystems, with decay. The subsystem involving the lasers (see Figure 6.14) has long since decayed because the relevant anomalies happened early in the race.

the planner software was correlated with these anomalies more-so than with the other sensors. As it turned out, there was an uninstrumented, shared component of the lasers that was causing this correlated behavior (see Chapter 4) 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 6.15, 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. Section 6.3.4 discusses the temporal properties of Stanley's bug in more detail.

Subsystems can describe global behavior as well as local behavior. Figure 6.16 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

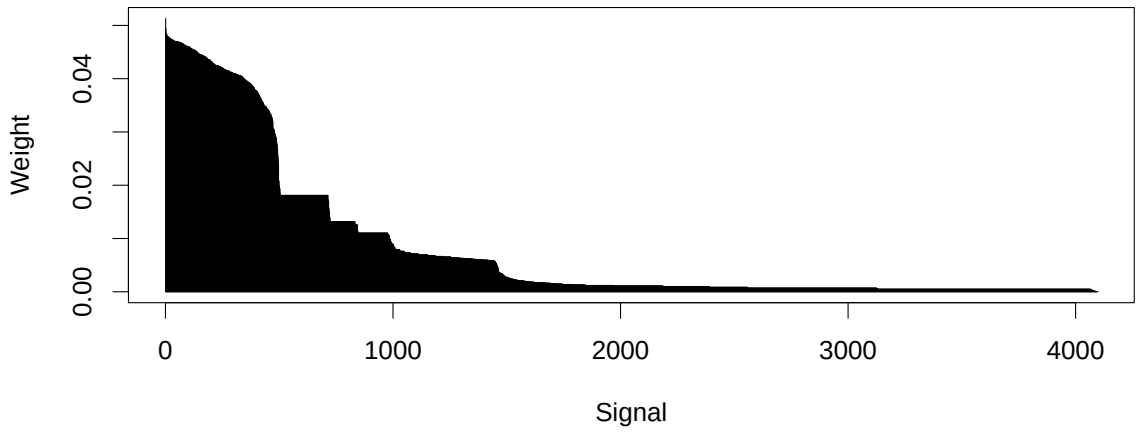



Figure 6.16: Weights of Spirit's first subsystem, sorted by weight magnitude. The compression stage has identified a phenomenon that affects many of the components.

(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 6.17, 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 (see Chapter 3). 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 root cause.

### 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

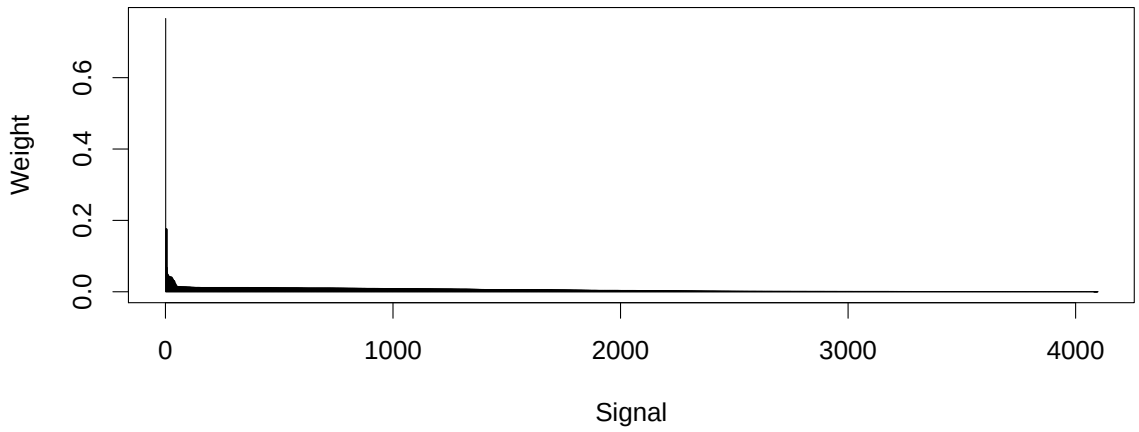



Figure 6.17: Sorted weights of Spirit's third subsystem. Most of the weight is in a small subset of the components.

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."

### 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 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

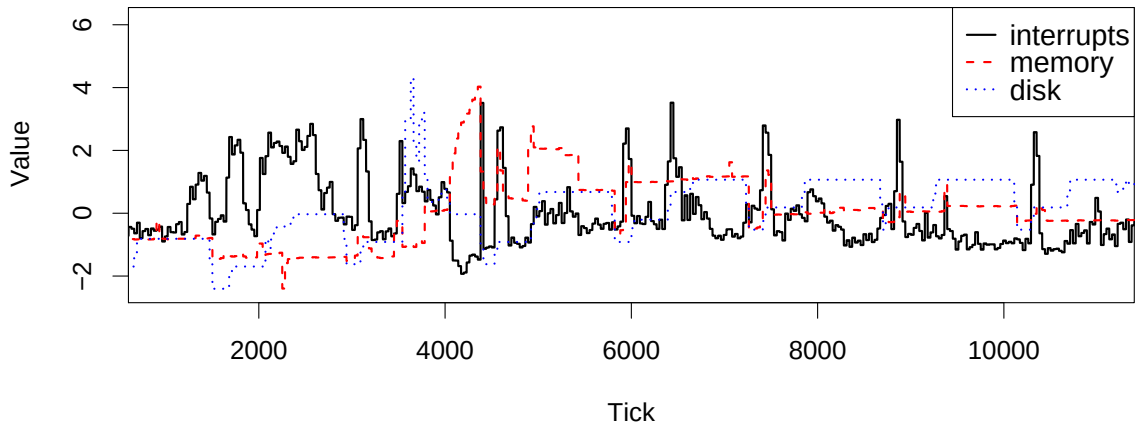



Figure 6.18: The anomaly signals of the representatives of the first three subsystems for the SQL cluster.

Figure 6.14, 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 6.18 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.

### 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

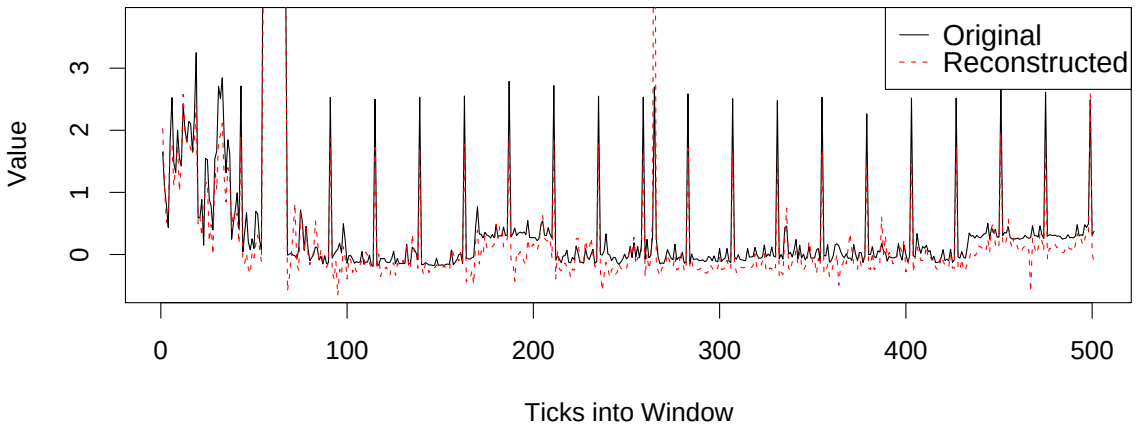



Figure 6.19: Reconstruction of a portion of Liberty's `admin` signal using the subsystems, including the periodic anomalies.

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 (see Section 4.4).

### 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 6.1.2). 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 6.19 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 6.20 shows the reconstruction of one of Liberty's indicator signals, with decay. The improvement in reconstruction accuracy

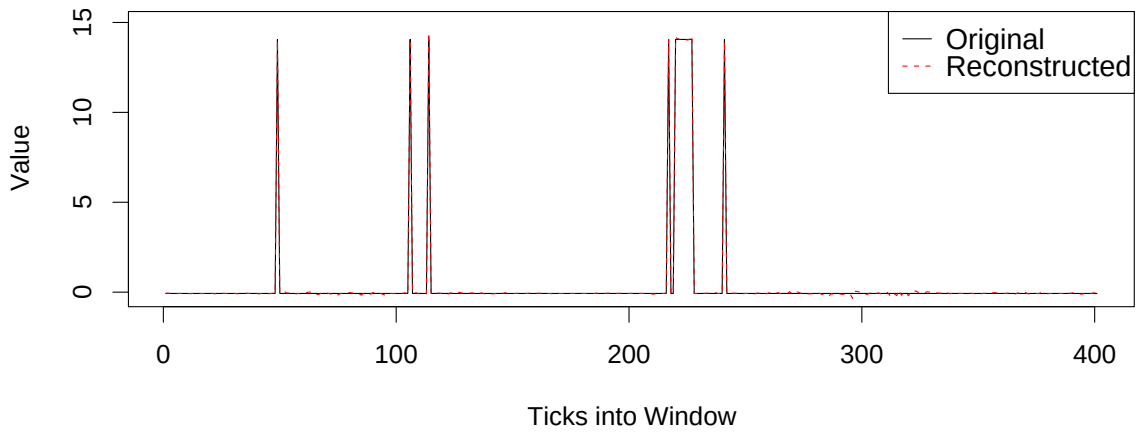



Figure 6.20: Reconstruction of a portion of Liberty's R_EXT_CCISS indicator signal with decay.

when using decay is apparent from Figure 6.21, which shows the relative reconstruction error for the SQL cluster. The behavior of this cluster changed near the end of the log due to an upgrade; the analysis with decay adapts to this change more easily.

### 6.3.4 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*.

#### Cascades

The logs were rich with instances of individual signals and behavioral subsystems with lag correlations. 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 6.3.3. This bug 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

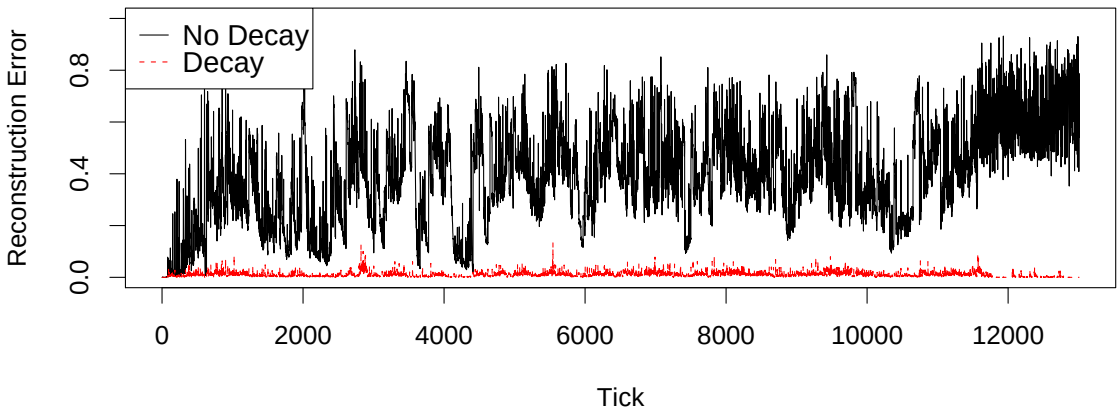



Figure 6.21: Relative reconstruction error for the SQL cluster, with and without decay. Reconstruction is more accurate when old values decay, especially during a new phase near the end of this log.

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; for instance, `LASER4` and `PLANNER_TRAJ` have a maximum correlation magnitude of 0.65 at a lag of 101 ticks.

In Section 6, 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 6.22). 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 toward a diagnosis.

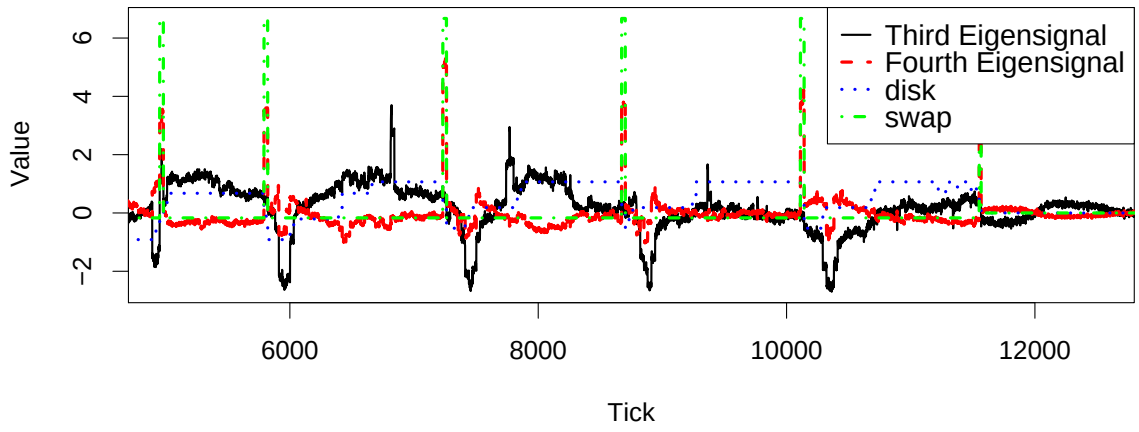



Figure 6.22: In the SQL cluster, the strongest lag correlation was found between the third and fourth subsystems, with a magnitude of 0.46 and delay of 30 minutes. These eigensignals and their representatives' signals (`disk` and `swap`, respectively), are shown above.

### Online Alarms

In addition to learning these cascades online, we can set alarms to trigger when the first sign of a cascade is detected, even when the cascade is already underway and even when we do not understand the underlying mechanism. 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.

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 6.23 and discussed briefly in Section 6): 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.

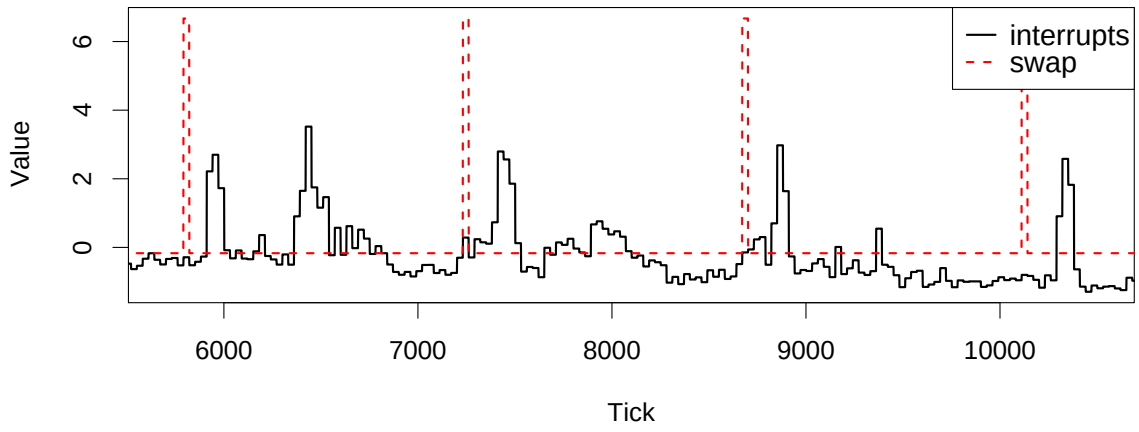



Figure 6.23: Our method reports that the signal `swap` tends to spike 210 minutes before `interrupts`, with a correlation of 0.271; we can detect this online.

#### 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.

### 6.3.5 Results Summary

Our results show that signal compression drastically increases the scalability of lag correlation (see Section 6.3.1) and that this compression process identifies behavioral subsystems with minimal information loss (see Section 6.3.2). Experiments on large

production systems (see Sections 6.3.3–6.3.4) 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.

## 6.4 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.

# Chapter 7

# Related Work

Work on log analysis and large-systems reliability has been hindered by a lack of data about their behavior. Schroeder [58] studied failures in a set of cluster systems at Los Alamos National Lab (LANL) using the entries in a *remedy database*. This database was designed to account for all node downtime in these systems, and was populated via a combination of automatic procedures and the extensive effort of a full-time LANL employee, whose job was to account for these failures and to assign them a cause within a short period of time after they happened. Schroeder also examined customer-generated disk replacement databases [59], but there was no investigation into how these replacements were manifested in the system logs. Although similar derived databases exist for the supercomputers studied in this thesis, our goal was to understand the behavior of the systems rather than human interpretations.

There is a series of papers on logs collected from Blue Gene/L (BG/L) systems. Liang, et al [27] studied the statistical properties of logs from an 8-rack prototype system, and explored the effects of spatio-temporal filtering algorithms. Subsequently, they studied prediction models [28] for logs collected from BG/L after its deployment at Lawrence Livermore National Labs (LLNL). The logs from that study are a subset of those used in this thesis. Furthermore, they identified alerts according to the *severity* field of messages. Although it is true that there exists a correlation between the value of the severity field of the message and the actual severity, we found

many messages with low severity that indicate a critical problem and vice versa. Section 2.1.2 elaborates on this claim and details the more intensive process we employed to identify alerts.

System logs for smaller systems have been studied for decades, focusing on statistical modeling and failure prediction. Tsao developed a *tuple* concept for data organization and to deal with multiple reports of single events [72]. Early work at Stanford [37] observed that failures tend to be preceded by an increased rate of non-fatal errors. Using real system data from two DEC VAX-cluster multicomputer systems, Iyer found that alerts tend to be correlated, and that this has a significant impact on the behavior and modeling of these systems [68]. Lee and Iyer [26] presented a study of software faults in systems running the fault-tolerant GUARDIAN90 operating system.

System logs are generally readily available and often contain critical clues to causes of failure, so many techniques for detecting alerts in logs have been proposed. Most prior work focuses on logs with dependable structure (easily tokenizeable into message-type-ID's). These attempts include pattern-learning [21], data mining techniques to discover trends and correlations [34, 62, 74, 78], and message timing [27].

Less work has been done in the area of unstructured message content. Attempts to apply techniques from genomic sequence mining to logs [62, 75] have run up against scaling problems. Vaarandi has applied clustering [73] and Apriori data mining [74], and was the first to encode word positions in his analyses (e.g., the first word of the message, the second, etc.), thereby effectively capturing a simple form of message context. Chapter 3 extends the understanding of how valuable such position encoding can be.

Reuning [50] and Liao [29] have each applied simple term weighting schemes to intrusion detection in logs, but Reuning concludes that his false positive rate renders the approach unusable in practice. In Chapter 3, we apply the more complex "log.entropy" weighting scheme that has been shown to be highly effective for information retrieval tasks [6].

There is an extensive body of work on system modeling, especially on inferring the causal or dependency structure of distributed systems. Our influence method

distinguishes itself from previous work in various ways, but primarily in that we look for *influences* rather than *dependencies* [3, 14, 60, 79]. Influence is an orthogonal property from dependencies that quantifies correlated deviations from normal behavior; influence is statistically robust to noisy or missing data and captures implicit interactions like resource contention.

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., [32, 61]). In our experience, it is often the case that none of these assumptions hold in practice. In contrast, our method requires no modifications to the system nor access to source code, does not require a specification of correct behavior nor predicates to check, and robustly handles the common case where not all components and their interactions are known.

One common thread in dependency modeling work is that the system must be actively perturbed by instrumentation [57] or by probing [7, 8, 12, 13, 53]. Pinpoint [10, 11] and Magpie [4] 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 [2] and WAP5 [52] use message traces and compute dependency paths (none of the systems we studied recorded such information). $D^3S$ [31] uses binary instrumentation to perform online predicate checks. Others leverage tight integration of the system with custom instrumentation to improve diagnosability (e.g., the P2 system [61]) or restrict the tool to particular kinds of systems (e.g., MapReduce [47] or wide area networks [17, 23, 24, 82]). Work by Bahl [3] aims to infer multi-level dependency graphs that model load-balancing and redundancy. Deterministic replay is another common approach [18, 32] but requires supporting instrumentation. For all of the production systems we studied, we could not apply any of these existing methods, and it was neither possible nor practical for us to add instrumentation. Indeed, the goal was sometimes to diagnose a bug that had already occurred; adding instrumentation would only help with future bugs. More generally, it may not be possible to modify existing instrumentation for reasons of system performance or cost.

Some approaches require the user to write predicates indicating what properties should be checked [31, 32, 61]. Pip [51] 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 [76], distributed diagnosis [19, 80, 81], and recovery [20]. Although our influence method 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.

With few exceptions [5], in previous work events are intrinsically binary (i.e., happen or not). Our approach, which abstracts components as real-valued signals, retains strictly more information about component behavior.

Many interesting problems in complex systems arise when components are connected or composed in ways not anticipated by their designers [35]. As systems grow in scale, the sparsity of instrumentation and complexity of interactions only increases. Our method infers a broad class of interactions using the existing instrumentation data and problem clues.

# Chapter 8

# Conclusions

This thesis examines the problem of understanding the interactions among components in complex production systems where the instrumentation data available for analysis may be noisy or incomplete. In particular, we consider the case where only some subset of the system components have instrumentation, and where that instrumentation does not completely describe the component state and may be heterogeneous across components.

## 8.1 Thesis Contributions

We conducted the largest-ever study of system logs, considering billions of log messages from five production supercomputers, and discussed the problem of identifying important messages called *alerts* in these data. Specifically, we described several of the key challenges: insufficient context, asymmetric problem reporting, system evolution, implicit correlation between messages, inconsistent message structure, and data corruption. Our analysis led us to recommend that systems be designed to log operational context (e.g., when periods of maintenance begin and end), that log filtering algorithms be aware of correlations between messages and root causes, that reliability be quantified based on meaningful values like useful work lost due to failures, and that failure prediction is a valuable and feasible future direction.

We presented the first reproducible results in alert detection and proposed an

information-theoretic algorithm, called Nodeinfo, that outperforms known techniques. Nodeinfo is based on the insight that similar computers executing similar workloads will tend to generate similar logs. One important contribution of this work was to formalize the problem of alert detection and propose specific metrics. The online version of this algorithm went into production use on several supercomputers at national labs.

We proposed a method for identifying the sources of problems in complex production systems where, due to the prohibitive costs of instrumentation, the data available for analysis may be noisy or incomplete. We defined *influence* as a class of component interactions that includes direct communication and resource contention. Our method infers the influences among components in a system by looking for pairs of components with time-correlated anomalous behavior. We summarize the strength and directionality of shared influences using a Structure-of-Influence Graph (SIG). Applications of this method to production systems showed that influence helps model systems and identify the likely causes of misbehavior.

Using influence, or correlated surprise, as the primitive, we created a query language for asking questions about component interactions and a tool called QI for efficiently answering them even for large systems. This language introduced the notion of metacomponents for representing aggregate behaviors, binary components for describing predicated behaviors, and masked components for hiding behaviors. We evaluated QI on real administrative tasks using system logs from four supercomputers, two autonomous vehicles, and a server cluster. The results showed that our tool can build models, isolate root causes, test hypotheses, and relate known problems to each other, even without modifying or perturbing the system under study.

We devised an efficient, two-stage, online method of computing influence that can analyze log data in real time for systems with hundreds of thousands of components. By virtue of being an online algorithm, we are able to use the output of our tool to identify cascading failures as they are underway and to set alarms that trigger in advance of misbehavior. We evaluated our method on real production systems and showed that it can produce operationally valuable results under conditions where other methods could not be applied: noisy, incomplete, heterogenous logs and systems

which cannot be modified or perturbed and for which we do not have full access to source code.

As systems trend toward more components and more sparse instrumentation, methods like ours—with only weak requirements on measurement data and good scaling properties—will become increasingly necessary for understanding system behavior.

# Bibliography


[1] N. R. Adiga and The BlueGene/L Team. An overview of the bluegene/l supercomputer. In *Supercomputing*, 2002.

[2] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Methitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, 2003.

[3] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *SIGCOMM*, 2007.

[4] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *OSDI*, 2004.

[5] Peter C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, 13(1):1–31, 1995.

[6] Michael W. Berry, Zlatko Drmac, and Elizabeth R. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Rev.*, 41(2):335–362, 1999.

[7] Mark Brodie, Irina Rish, and Sheng Ma. Optimizing probe selection for fault localization. In *Workshop on Distributed Systems: Operations and Management (DSOM)*, 2001.

[8] 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.

[9] M. F. Buckley and D. P. Siewiorek. A comparative analysis of event tupling schemes. In *FTCS-26, Intl. Symp. on Fault Tolerant Computing*, pages 294–303, June 1996.

[10] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. Path-based failure and evolution management. In *NSDI*, 2004.

[11] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: problem determination in large, dynamic internet services. In *DSN*, 2002.

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

[13] Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP*, 2005.

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

[15] Dror G. Feitelson and Dan Tsafrir. Workload sanitation for performance evaluation. In *IEEE Intl. Symp. Performance Analysis Syst. & Software (ISPASS)*, pages 221–230, Mar 2006.

[16] Ulrich Flegel. Pseudonymizing unix log files. In *Proceedings of the Infrastructure Security Conference (InfraSec)*, 2002.

[17] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-Trace: A pervasive network tracing framework. In *NSDI*, 2007.

[18] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *USENIX Technical*, 2006.

[19] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *SOSP*, 2009.

[20] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A framework for cloud recovery testing. In *NSDI*, 2011.

[21] J. L. Hellerstein, S. Ma, and C.S. Perng. Discovering actionable patterns in event data. *IBM Systems Journal*, 41(3), 2002.

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

[23] Srikanth Kandula, Dina Katabi, and Jean-Philippe Vasseur. Shrink: A tool for failure diagnosis in IP networks. In *MineNet Workshop at SIGCOMM*, 2005.

[24] Ramana Rao Kompella, Jennifer Yates, Albert Greenberg, and Alex C. Snoeren. IP fault localization via risk modeling. In *NSDI*, 2005.

[25] S. Kullback. The Kullback-Leibler distance. *The American Statistician*, 41:340–341, 1987.

[26] I. Lee and R. K. Iyer. Faults, symptoms, and software fault tolerance in the tandem guardian90 operating system. In *Fault-Tolerant Computing. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 20–29, 1993.

[27] Y. Liang, Y. Zhang, A. Sivasubramaniam, R. K. Sahoo, J. Moreira, and M. Gupta. Filtering failure logs for a bluegene/l prototype. In *DSN*, 2005.

[28] Yinglung Liang, Yanyong Zhang, Morris Jette, Anand Sivasubramaniam, and Ramendra K. Sahoo. Blue gene/l failure analysis and prediction models. In *DSN*, 2006.

[29] Yihua Liao and V. Rao Vemuri. Using text categorization techniques for intrusion detection. In *11th USENIX Security Symposium, August 5–9, 2002.*, pages 51–59, 2002.

[30] T. T. Y. Lin and D. P. Siewiorek. Error log analysis: statistical modeling and heuristic trend analysis. *Reliability, IEEE Transactions on*, 39(4):419–432, 1990.

[31] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: debugging deployed distributed systems. In *NSDI*, 2008.

[32] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS Checker: Combating bugs in distributed systems. In *NSDI*, 2007.

[33] Logsurfer. *A tool for real-time monitoring of text-based logfiles.* http://www.cert.dfn.de/eng/logsurf/, 2006.

[34] S. Ma and J. Hellerstein. Mining partially periodic event patterns with unknown periods. In *ICDE*, 2001.

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

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

[37] F. A. Nassar and D. M. Andrews. A methodology for analysis of failure prediction data. In *Real-Time Systems Symposium*, pages 160–166, December 1985.

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

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

[40] A. J. Oliner, L. Rudolph, and R. Sahoo. Cooperative checkpointing theory. In *IPDPS*, 2006.

[41] A. J. Oliner, L. Rudolph, and R. K. Sahoo. Cooperative checkpointing: A robust approach to large-scale systems reliability. In *ICS*, June 2006.

[42] A. J. Oliner, L. Rudolph, R. K. Sahoo, J. E. Moreira, and M. Gupta. Probabilistic qos guarantees for supercomputing systems. In *DSN*, 2005.

[43] A. J. Oliner, R. K. Sahoo, J. E. Moreira, M. Gupta, and A. Sivasubramaniam. Fault-aware job scheduling for bluegene/l systems. In *IPDPS*, 2004.

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

[45] Adam J. Oliner and Alex Aiken. Online detection of multi-component interactions in production systems. In *DSN*, 2011.

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

[47] Xinghao Pan, Jiaqi Tan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Ganesha: Black-box fault diagnosis for MapReduce systems. Technical report, CMU-PDL-08-112, 2008.

[48] Spiros Papadimitriou, Jimeng Sun, and Christos Faloutsos. Streaming pattern discovery in multiple time-series. In *VLDB*, 2005.

[49] Jim Prewett. Analyzing cluster log files using logsurfer. In *Proceedings of the 4th Annual Conference on Linux Clusters*, 2003.

[50] John R. Reuning. Applying term weight techniques to event log analysis for intrusion detection. Master's thesis, University of North Carolina at Chapel Hill, July 2004.

[51] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, 2006.

[52] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Marcos K. Aguilera, and Amin Vahdat. WAP5: black-box performance debugging for wide-area systems. In *WWW*, 2006.

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

[54] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *KDD*, 2003.

[55] R. K. Sahoo, A. Sivasubramanian, M. S. Squillante, and Y. Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *DSN*, 2004.

[56] Yasushi Sakurai, Spiros Papadimitriou, and Christos Faloutsos. BRAID: Stream mining through group lag correlations. In *SIGMOD*, 2005.

[57] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *NSDI*, 2011.

[58] Bianca Schroeder and Garth Gibson. A large-scale study of failures in high-performance-computing systems. In *DSN*, June 2006.

[59] Bianca Schroeder and Garth Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you? In *FAST*, 2007.

[60] Reinhard Schwarz and Friedmann Mettern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing*, 7(3):149–174, March 1994.

[61] Atul Singh, Petros Maniatis, Timothy Roscoe, and Peter Druschel. Using queries for distributed monitoring and forensics. In *EuroSys*, 2006.

[62] J. Stearley. Towards informatic analysis of syslogs. In *IEEE International Conference on Cluster Computing*, pages 309–318, 2004.

[63] Jon Stearley. Defining and measuring supercomputer Reliability, Availability, and Serviceability (RAS). In *Proceedings of the Linux Clusters Institute Conference*, 2005. See http://www.cs.sandia.gov/~jrstear/ras.

[64] Jon Stearley. *Scrubbed logs from five top supercomputers.* http://www.cs.sandia.gov/∼jrstear/.logs-alpha1, 2008.

[65] Jon Stearley. *Sisyphus—a log data mining toolkit.* http://www.cs.sandia.gov/sisyphus, 2008.

[66] Jon Stearley and Adam J. Oliner. Bad words: Finding faults in spirit's syslogs. In *Workshop on Resiliency in High-Performance Computing (Resilience)*, 2008.

[67] H. A. Sturges. The choice of a class interval. *J. American Statistical Association*, 1926.

[68] D. Tang and R. K. Iyer. Analysis and modeling of correlated failures in multicomputer systems. *Computers, IEEE Transactions on*, 41(5):567–577, 1992.

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

[70] S. Thrun and M. Montemerlo, et al. Stanley: The robot that won the DARPA Grand Challenge. *Journal of Field Robotics*, 23(9):661–692, June 2006.

[71] Top500. *Top500 Supercomputing Sites.* http://www.top500.org/, June 2006.

[72] M. M. Tsao. *Trend Analysis and Fault Prediction.* PhD dissertation, Carnegie-Mellon University, May 1983.

[73] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of IEEE International Workshop on IP Operations and Management (IPOM)*, pages 119–126, October 2003.

[74] Risto Vaarandi. A breadth-first algorithm for mining frequent patterns from event logs. In *Proceedings of the 2004 IFIP International Conference on Intelligence in Communication Systems*, volume 3283, pages 293–308, 2004.

[75] A. Wespi, M. Dacier, and H. Debar. An intrusion-detection system based on the teiresias pattern-discovery algorithm. In *EICAR Annual Conference Proceedings*, pages 1–15, 1999.

[76] Wei Xu, Ling Huang, Armando Fox, Dave Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP*, 2009.

[77] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Online system problem detection by mining patterns of console logs. In *ICDM*, 2009.

[78] Kenji Yamanishi and Yuko Maruyama. Dynamic syslog mining for network failure monitoring. In *KDD*, 2005.

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

[80] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. SherLog: Error diagnosis by connecting clues from run-time logs. In *ASPLOS*, 2010.

[81] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. In *ASPLOS*, 2011.

[82] Yao Zhao, Zhaosheng Zhu, Yan Chen, Dan Pei, and Jia Wang. Towards efficient large-scale VPN monitoring and diagnosis under operational constraints. In *INFOCOM*, 2009.