A Query Language for Understanding Component Interactions in Production Systems

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

ABSTRACT

When something unexpected happens in a large production system, administrators must first perform a search to isolate which components and component interactions are likely to be involved. The system may consist of thousands of interacting subsystems, the logging instrumentation may be noisy or incomplete, and the problem description may be vague, so this search is often the most difficult part of understanding the system's behavior. To facilitate the search process, we present a query language and a method for computing these queries that makes minimal assumptions about the available data. We evaluate our method on nearly 1.22 billion lines of system logs from four supercomputers, two autonomous vehicles, and a server cluster.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management; H.2.3 [**Information Systems**]: Query Languages

General Terms

Management, Reliability, Languages, Algorithms

Keywords

Query language, logs, influence, correlation, production systems

1. INTRODUCTION

When something unexpected happens in a large production system —a program crashes, a node's performance flags, a power supply overheats—administrators face several problems at once. First, they may be unable to describe the event any more accurately than the approximate time it occurred. Second, they must diagnose the problem using only the data that was recorded when the issue manifested (primarily log files); this data may be noisy and may not describe all components and their interactions. Third, the system may have many components (tens to thousands), and the administrators must identify which components and component interactions are likely to have been involved.

Consider the following example. Users notice that their jobs are failing more frequently. The typical process for a system administrator is to search the job logs to figure out what components were used by these jobs, scour the system logs from those components for any messages that might hint at a cause, and possibly expand the search to other related components based on their expert knowledge of the system. The key observation is that this is fundamentally a search problem—one for which the state-of-practice is primarily manual, tedious, and ad hoc—where the administrator asks, "What components and interactions are likely to be involved with these job failures?" The input to the search is the available measurements from instrumentation and a simple description of the behavior we wish to understand; the goal of the search is to identify the components and interactions that are likely to be involved.

In this paper we present a method for using simple user specifications of when and where a problem manifested, together with existing instrumentation, to compute the components and interactions that are likely to be involved with the problem. Our method computes which system components statistically influence the behavior of other components and which components are statistically linked with the problem. Influences are represented as a *Structure-of-Influence Graph* (SIG), where the nodes are components and the problem we are trying to understand, and influences are edges between nodes. A discussion of computing SIGs is covered elsewhere [20].

Continuing the example above, the administrator would specify that surprising behavior was observed around the times the users' jobs failed. Using that clue alone, our system QI (for Querying Influence; pronounced "CHē") determines what other components deviated from normal behavior around those times and generates a SIG to summarize the correlations (see Section 3). For example, QI might generate the hypothetical SIG in Figure 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, 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

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

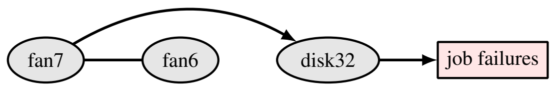

Figure 1: An example SIG showing a chain of influence related to the job failures.

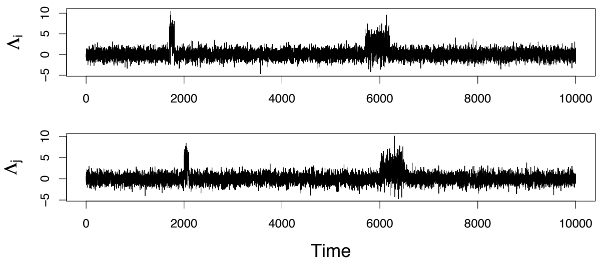

Figure 2: Anomaly signals for the hypothetical components `fan7` (Λ_i) and `disk32` (Λ_j). By inspection, the signals look similar; our method mathematically describes this similarity.

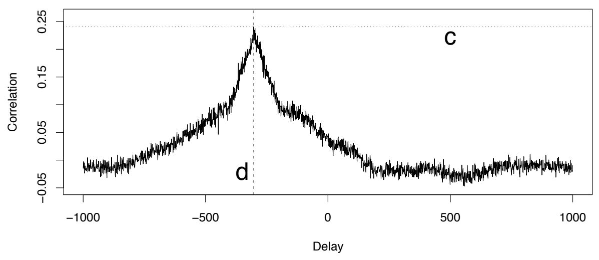

Figure 3: Cross-correlation between the anomaly signals of `fan7` and `disk32`. The peak at delay d has height (correlation) c, indicating that surprising behavior on `fan7` tends to be followed by surprising behavior on `disk32` with a lag of d.

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 paper, we present QI's query language (see Section 2) and our implementation of an infrastructure for efficiently computing these queries (see Section 3). In Section 4, we describe data sets collected from seven different systems: four supercomputers, one server cluster, and two embedded, distributed systems. Following the lead of previous work in this area [16, 27], we primarily evaluate our method with case studies on these data sets; Section 5 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.

2. 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*. The choice of method for converting raw measurements (logs) into anomaly signals is important but not the subject of this paper. The field of anomaly detection has proposed many options, and we have found that even simple algorithms give useful results. For example, the *nodeinfo* algorithm [19], which we use for unstructured textual logs (see Sections 4.1 and 4.2), raises the value of the signal when the distribution of terms in a recent window of log messages is unlike the historical distribution.

Consider the hypothetical components `fan7` and `disk32` from Figure 1, whose anomaly signals are plotted in Figure 2. These two signals have similar structure, 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.

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 [20]. The SIG in Figure 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 2.1 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 2.2 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 pair-wise relationships between components in focus. However, the user can also 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 2.3.

2.1 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 [20]. We explain these computations using the example from Section 1, by describing how QI infers the directed edge from `fan7` to `disk32`.

To determine whether unusual behavior on `fan7` (represented by the anomaly signal Λ_i) correlates in time with unusual behavior on `disk32` (Λ_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 μ_i and σ_i are the mean and standard deviation of Λ_i, respectively. Figure 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.

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

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

where $\mathrm{argmax}_t f(t)$ 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 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 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 ε specified—implicitly or explicitly—by the query. The edge is directed because the corresponding delay d lies outside of some noise margin α. 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. This was not an issue on any of the systems we studied (see Section 4).

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.

2.2 Query Examples

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

2.2.1 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
```

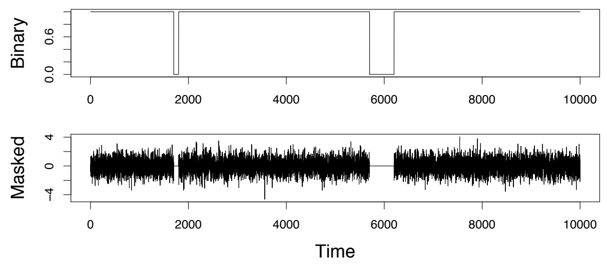

Figure 4: Applying the binary component (top) as a mask on `fan7` yields the masked component (bottom), whose masked values are replaced by the average of the unmasked values.

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 5.1, we use Qi in a similar way to understand job failures on a supercomputer.

2.2.2 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 5.1 to identify a data corruption problem and in Section 5.6 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. The keyword `all` is shorthand for every component currently in the system. The `edge` parameter sets the ε threshold (see Section 2.1) so that only correlations stronger than 0.25 appear as edges in the SIG; the `arrow` parameter sets the α 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 5, we construct binary components whose anomaly signals are high exclusively in intervals where the corresponding logs match a regular expression.

2.2.3 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 5.3). 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 4, apply it to `fan7` (Figure 2, top), and get the masked component on the bottom in Figure 4.

2.3 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 2.2.1). The set of component types may be extended. An interval (e.g., `a:b`) can be used as a `<range>` (see Section 2.2.2) or a `<mask>` (the optional `!` inverts the mask; see Section 2.2.3). The `all` keyword represents every component in the system and `last` stands for every component that appeared in the previously plotted SIG.

3. QI IMPLEMENTATION

We have implemented the query language in Section 2 as a tool called QI, written in Python.

QI spends the majority of the time computing cross-correlations and finding local extrema (see Section 2.1). 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 5.7). 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 $\mathbf{D}$.

4. SYSTEMS

We evaluate QI using data from seven production systems: four supercomputers, one cluster, and two autonomous vehicles. Table 1 gives a summary of these systems and logs, described in Sections 4.1–4.4 and elsewhere [18, 19, 20, 21, 28, 29]. 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 way for our experiments.

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 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.

4.1 Supercomputers

We use four publicly available logs from supercomputers that were in production use at national laboratories [28]. These four systems, named Liberty, Spirit, Thunderbird, and Blue Gene/L (BG/L), vary in size by several orders of magnitude, ranging from 512 processors in Liberty to 131,072 processors in BG/L. The logs were recorded during production use of these systems and we make no modifications to them, whatsoever. An extensive study of these logs can be found elsewhere [21]. We use the so-called *nodeinfo algorithm*, which is based on the frequency of terms appearing in log messages [19], to generate anomaly signals from the raw data.

4.2 Mail-Routing Cluster

We also obtained logs from 17 machines of a Stanford University campus email routing server cluster. Of these servers, 16 recorded two types of logs: a typical mail server log (denoted `mail`) and a Pure Message log (a spam and virus filtering application, denoted `pmx_log`). One system recorded only the mail log. The nodeinfo algorithm works on unstructured textual data, so, as in Section 4.1, we apply it to generate anomaly signals for these logs.

4.3 Autonomous Vehicles

Stanley is the autonomous diesel-powered Volkswagen Touareg R5 developed at Stanford University that won the DARPA Grand Challenge in 2003 [29]. A modified 2006 Volkswagen Passat wagon named Junior placed second in the subsequent Urban Challenge [18]. 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 (IPC). These log messages indicate their source, but not their destination (there are sometimes multiple consumers). We use the actual, raw logs from the Grand Challenge and Urban Challenge, respectively, and compute anomaly signals from these data using an existing method [20].

4.4 Log Contents

Table 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 rou-

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 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 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.

tine 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 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. 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 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 5.7 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 paper, 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).

5.1 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 2.2.1). 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 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

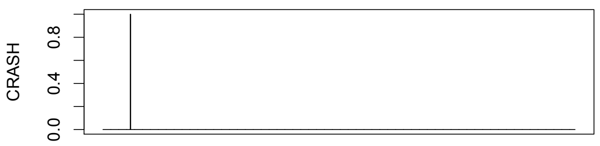

Figure 5: The anomaly signal of the synthetic `CRASH` component, which encodes when a job on BG/L crashed.

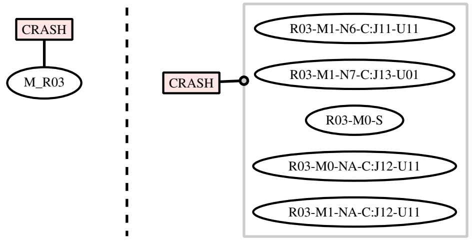

Figure 6: Components most strongly correlated with the crash. On the left, the metacomponent; on the right, the components within that metacomponent.

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 shows the anomaly signal for this synthetic component. The result of the query is on the left in Figure 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 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 [19, 21]: `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.

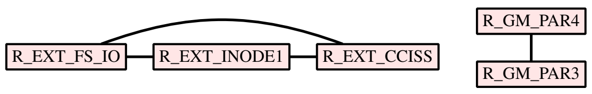

Figure 7: Some alert types on Liberty are correlated; QI helps search for the reasons why.

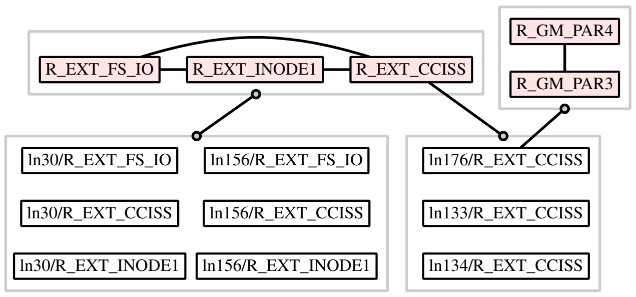

Figure 8: Synthetic components can show the relationships among both when certain alert types are generated and when they are generated on individual components.

5.2 Correlated Alerts on Liberty

Given an alert message, such as the one in Section 5.1, 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 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 [21], 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 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 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 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.

5.3 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

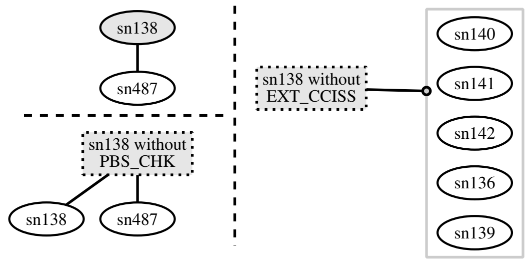

Figure 9: Masking the contribution of one anomaly source can make other shared influences more apparent.

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 9 shows the result: there is exactly one other node that seems correlated with `sn138`. Based on previous work on this log [21], 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)

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 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 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 our query—between `sn138` and other components.

5.4 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, 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 [21]. 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.

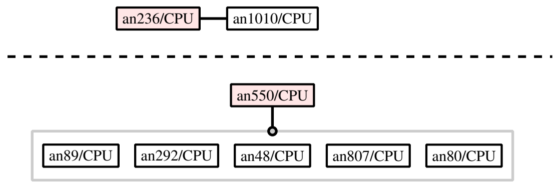

Figure 10: Binary components representing the 'CPU' alert tend to share a strong influence with sets of such components that are topologically close, such as those in the same job scheduling group.

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 5.2. 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 10, `an236/CPU` shares a strong influence with `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 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 (omitted for space reasons), 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.

5.5 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 4.2 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 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.

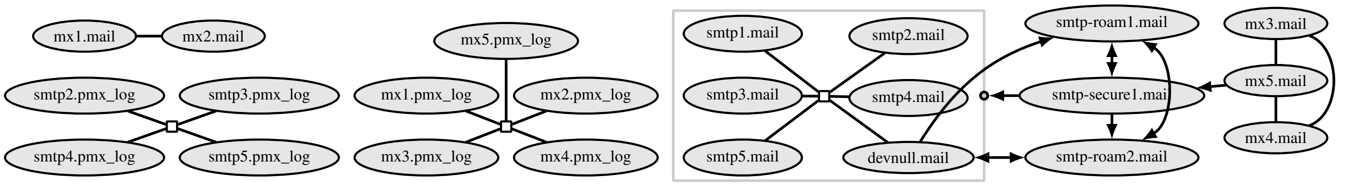

Figure 11: Shared influence in a cluster of mail-routing servers.

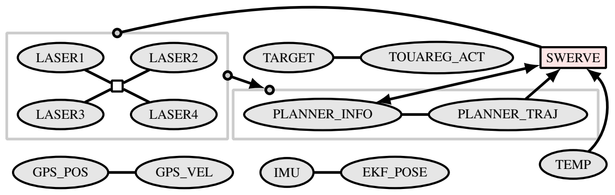

Figure 12: A complete SIG for Stanley, including a synthetic component, `SWERVE`, that represents an unexpected behavior.

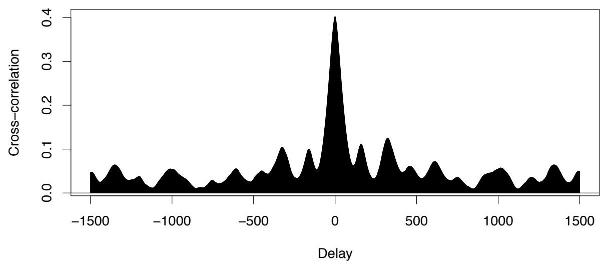

Figure 13: Cross-correlation between `LASER1` and `LASER2`.

5.6 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 detail elsewhere [20, 29], 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, previous work explained how to use SIGs in an ad hoc way to isolate a shared component of the lasers as a likely cause [20].

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 the original paper (including the same edge and arrow thresholds), shown in Figure 12, using the following command:

`arrow=90 edge=0.15 all SWERVE<-60:100.`

Figure 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 (we omit these results for space reasons).

Although Figure 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 really intends to

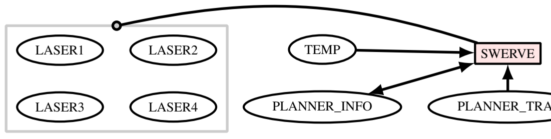

Figure 14: The components most strongly correlated with the swerving behavior.

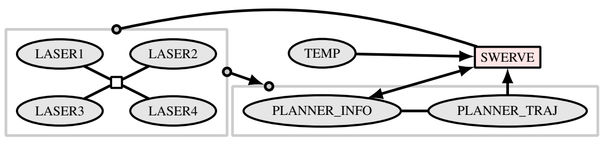

Figure 15: This automatically generated graph implies that a shared component of the lasers is likely to be causally related to the swerving.

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 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 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 [20].

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).

5.7 Performance and Scaling

We have used QI with systems containing as many as 69,087 log-generating components (see Section 5.1). 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

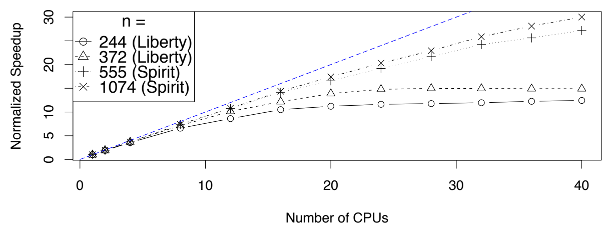

Figure 16: Speedups for components on Liberty and Spirit, normalized according to the runtime on a single local CPU. Those baselines were 117.31 sec, 276.18 sec, 1208.64 sec, and 4611.76 sec, respectively.

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 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 5.1–5.6, the summary of runtimes in Table 3 shows that QI can be used interactively.

6. RELATED WORK

There is an extensive body of work on system modeling, especially on inferring the causal or dependency structure of distributed systems. Our method distinguishes itself from previous work in various ways, but primarily in that we look for *influences* [20] rather than *dependencies* [2, 10, 26, 31]. 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. These low-level properties have been shown using controlled experiments on idealized systems [20]; the focus of this paper is on providing a high-level and systematic method for applying these techniques to real systems.

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., [16, 27]). 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 or by probing [4, 5, 8, 9, 25]. Pinpoint [6, 7] and Magpie [3] track communication dependencies with the aim of isolating the root cause of misbehavior; they require instrumentation of the application to tag client requests. In order to determine the causal relationships among messages, Project5 [1] and WAP5 [24] use message traces and compute dependency paths (none of the systems we studied recorded such information). D^{3}S [15] 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 [27]) or restrict the tool to particular kinds of systems (e.g., MapReduce [22] or wide area networks [13, 14]). Deterministic replay is another common approach [11, 16] but requires supporting instrumentation. For all seven of the production systems we study, we could not apply any of these existing methods, and it was neither possible nor practical for us to add instrumentation. 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 also require the user to write predicates that indicate what properties should be checked [15, 16, 27]. Pip [23] identifies when communication patterns differ from expectations and requires an explicit specification of those expectations. We have no such predicates, models, or specifications for any of the systems we study. Furthermore, for several of the bugs our method isolates, it would not have been possible to write a sufficient specification of correct behavior before diagnosing the bugs—in other words, knowing what property to check (e.g., creating a model suitable for model checking) was equivalent to understanding the root cause of the bug (see Sections 5.4 and 5.6).

Recent work shows how access to source code can facilitate tasks like log analysis [30] and distributed diagnosis [12]. Although our system could be extended to take advantage of such access, many systems involve proprietary, third-party, or classified software for which source code is unavailable.

Many interesting problems in complex systems arise when components are connected or composed in ways not anticipated by their designers [17]. 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.

7. 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). Using raw data from seven unmodified production systems, we demonstrated the use of QI for such tasks as alert discovery (see Section 5.1), problem isolation (see Sections 5.3, 5.4, and 5.6), and general system and interaction modeling (see Sections 5.2 and 5.5). As we demonstrated, our method scales linearly with system size and is fast enough to be used interactively for typical use cases (see Section 5.7). 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.

Acknowledgments

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

8. REFERENCES

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Methitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, pages 74–89, 2003.

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

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-Routing Cluster	`all`	2.36	528
Stanley	`periph=all SWERVE<-60:100`	20.37	16
	`last`	35.60	120

Table 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.

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

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

[5] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *IEEE IM*, pages 377–390, Seattle, WA, 2001.

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

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

[8] S. Chutani and H. Nussbaumer. On the distributed fault diagnosis of computer networks. In *IEEE Symposium on Computers and Communications*, pages 71–77, Alexandria, Egypt, June 1995.

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

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

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

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

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

[14] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. IP fault localization via risk modeling. In *NSDI*, pages 57–70, 2005.

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

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

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

[18] M. Montemerlo et al. Junior: The Stanford entry in the Urban Challenge. *Journal of Field Robotics*, 25(9):569–597, 2008.

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

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

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

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

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

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

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

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

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

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

[29] 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.

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

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