Performance Debugging in Data Centers: Doing *More* with *Less*[1]

Emmanuel Cecchet[†], Maitreya Natu[‡], Vaishali Sadaphal[‡], Prashant Shenoy[†], Harrick Vin[‡]

[†]*Computer Science Department, University of Massachusetts at Amherst, USA*
[‡]*Tata Research Development and Design Centre (TRDDC), Tata Consultancy Services, Pune, India*

***Abstract* — With the increasing scale and complexity of data centers, detecting and localizing performance faults in real-time has become both a pressing need and a challenge. While several approaches for performance debugging in data centers have been proposed, these techniques do not assume any constraints on the availability of operational data needed to detect and localize faults. We argue that collecting such operational data often requires significant instrumentation or intrusiveness, which is difficult to realize in production data centers. Such constraints complicate the deployment of existing techniques or limit their effectiveness in practice. In this paper, we argue that for performance debugging to become practical and effective in real-world systems, one needs to develop techniques that are "*more* effective" *with* "*less* instrumentation and intrusiveness". We raise several issues and challenges in realizing this vision and present some initial ideas on addressing these challenges.**

***Index Terms*—data centers, performance debugging, fault detection and localization, operating and distributed systems.**

I. Introduction

A defining characteristic of the information age is our reliance on data centers—consisting of large numbers of computing, communication, and storage systems as well as wide-range of applications and services. The scale and complexity of these data centers, however, have been increasing rapidly. This, in turn, is limiting our ability to *understand* and *control* the operations of such data centers.

Consider, for instance, an equity trading plant operated by a top-tier investment bank in the US. This data center receives and processes 4-6 millions of requests for equity trades (referred to as *orders*) and 10-100 million *market updates* (news, stock-tick updates, etc.) each day. Upon its arrival, each order goes through several processing steps prior to being dispatched to a stock exchange (e.g., New York Stock Exchange (NYSE)) for execution. Similarly, market updates are processed, enriched, aggregated and then transmitted to thousands of program trading engines as well as traders' workstations. The IT infrastructure for processing these orders and market updates consists of thousands of application components running on several hundred servers. Orders and market updates hop from one component/server to another prior to reaching their destination. Figure 1 depicts a portion of the data center operated by this investment bank for processing trading orders; each node in this graph represents an application process and edges indicate the flow of requests from one process to next.

A critical business requirement in this environment is that the end-to-end latency for processing each *request* should not exceed 7-10*ms*. In the event that end-to-end delay starts to exceed this threshold consistently, one needs to detect, localize and correct the "*fault*" rapidly (*near-real-time*). Note that end-to-end delay may increase because of dynamic changes in workload (leading to congestion at a processing node) or slowing down of a processing node because of hardware or software errors (e.g., memory leak). The longer it takes to detect and localize faults, the greater is the business impact. Today, unfortunately, because of the scale and complexity of data centers, the volume of requests, and the manual analysis processes, localizing performance faults often takes hours (and at times, even days).

Detecting and localizing performance faults[2]—referred to as *performance debugging*—in such data centers involve four key steps:

1. Build a *model* of *normal operations* of a system (generally through off-line analysis of data obtained by instrumenting the system);
2. Place probes to monitor the operational system;
3. Detect performance faults in near-real-time; and
4. Localize faults by combining the knowledge (model and monitored data) derived in previous steps.

[1] This research was supported in part by NSF grants CNS-0720271 and CNS-0720616. Authors are listed in alphabetical order and can be contacted at *{cecchet,shenoy}@cs.umass.edu* and *{maitreya.natu,vaishali.sadaphal, harrick .vin}@tcs.com*

[2] In this paper, we only consider faults that impact performance (e.g., end-to-end latency or throughput) of the system, and not fail-stop faults.

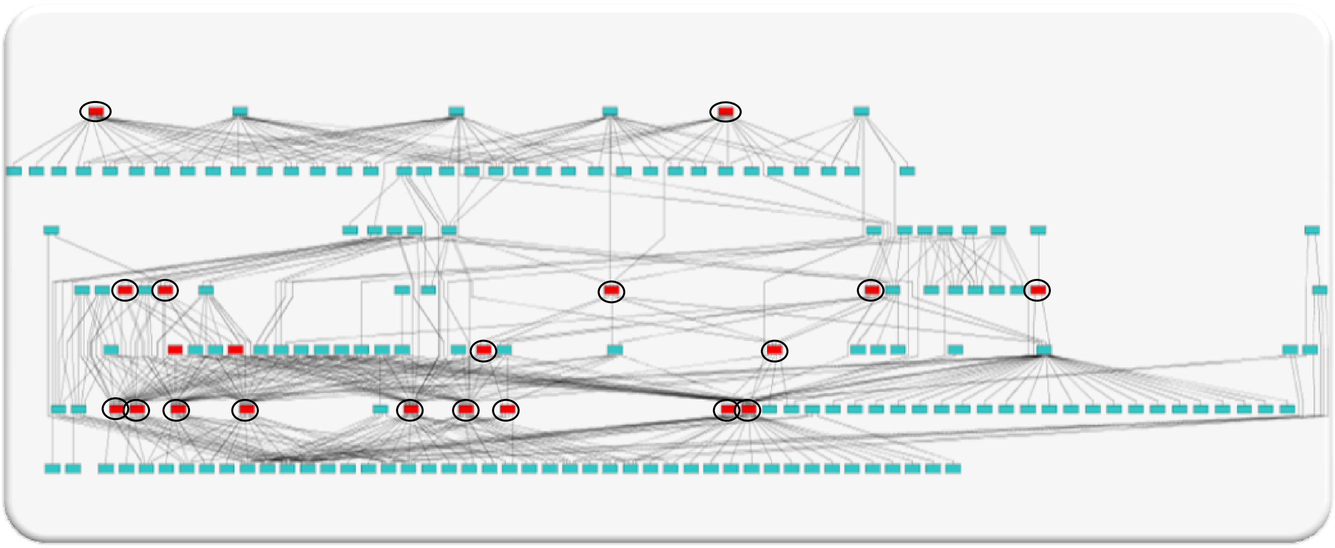


Figure 1. Structure of an Equity Trading Application at a Top-tier Investment Bank. Circled nodes are the top-k application components in terms of the workload processed by each node in the system.

The effectiveness of the above steps depends on the number and the type of data collection *probes* available in the system. Retrofitting an operational system with the instrumentation required to facilitate performance debugging, however, is always a challenge. System operators and administrators are reluctant to introduce probes into the production environment, especially if the probes are intrusive (and can modify the system behavior). Thus, a key practical requirement is that *a performance debugging solution should minimize the amount of* instrumentation *needed to gather real-time operational statistics and the* intrusiveness *of these data gathering methods*.

Much of the prior research in the area of performance debugging has ignored this very basic *practical* requirement. Further, much of the prior work has focused on *algorithms* for fault localization, while assuming that all of the data required by the algorithm for its decision making can be easily gathered. Unfortunately, in most cases, collecting such data either requires significant *instrumentation* (e.g., requiring a probe to be placed at each process or server) or *intrusiveness* (e.g., requiring that each request carries a request-ID end-to-end such that the debugging system can track flow of requests). This makes these techniques difficult to deploy in real-world operational systems.

In this paper, we argue that for automated performance debugging to become practical and effective in real-world systems, one needs to develop techniques that are *more* effective even *with less* instrumentation and intrusiveness. Our goal here is to raise several issues and challenges in designing these techniques – rather than propose solutions for specific settings.

The rest of the paper is organized as follows. In Section II, we first argue that effective performance debugging can be achieved by using *either* significant instrumentation *or* intrusiveness but is challenging when constraints are placed on both. We then raise the *"more with less"* question: Is it possible to achieve effective performance debugging using low instrumentation *and* low intrusiveness? In Section III, we discuss an example scenario with low instrumentation and intrusiveness, and describe a straw man approach for performance debugging in this environment. The straw man approach appears promising; however, it has several limitations. We use this discussion to derive, in Section IV, a set of research challenges for developing practical and effective performance debugging solutions. Finally, Section V concludes our paper.

II. INSTRUMENTATION AND INTRUSIVENESS

Performance debugging in data center requires two different types of instrumentation: (1) to detect faults in near real-time; and (2) to build an operational model of the system for subsequent fault localization. Interestingly, the amount of instrumentation and intrusiveness required for one task is generally quite different from what is needed for the other.

A. *Instrumentation for Fault Detection*

The instrumentation required for fault detection depends on the primary performance metric of the application (e.g., end-to-end latency or throughput). For instance, if the primary performance metric is end-to-end latency, then the instrumentation must timestamp each request upon arrival into and departure from the system, and take the difference between the two timestamps. Since each request must carry the arrival timestamp with it, the required instrumentation is inherently intrusive.

If, on the other hand, throughput is the primary performance metric for an application, then the instrumentation for fault detection simply needs to compute the number of requests departing the system within a defined interval. This can be done with very little intrusiveness. These insights yield our first observation:

Observation 1: The instrumentation intrusiveness is a direct function of the performance metric of interest.

B. Instrumentation for Fault Localization

The goal of fault localization is to identify the component (process, server, or workload) that is the root-cause for the performance fault.

A simple solution consists of measuring performance metrics of interest as well as resource utilization levels at *all* servers in a data center. A fault can be then be localized by detecting significant changes in the measured values at servers. An example of this approach is the work by Cohen et al. [8] that involves monitoring resource usage of all servers, and then correlating SLA violations to resource usage on individual servers.

However, this simple approach requires a large amount of instrumentation and incurs significant overhead—both in terms of monitoring data volume and run-time overhead. Additionally, if the instrumentation yields per-server measurements, as opposed to end-to-end measurements, then we may see several false positives with respect to fault detection since a significant change in performance metrics at a node (e.g., per-hop delay) may not lead to significant changes in end-to-end performance metrics (e.g., relationship between end-to-end delay and SLA).

More sophisticated solutions, on the other hand, involve collecting information about the operational semantics of the system, such as the interactions between different components of the system (referred to as the per-hop dependencies) or the flow of requests through the system components (namely, the identification of end-to-end paths for each request). This type of inter-component dependency information allows a performance debugging system to localize faults without requiring each node to be instrumented. Note, however, that extraction of such operational semantics and dependencies require much greater level of “intrusiveness” because such monitors require modifications at the system, middleware, or application levels. Further, different types of monitors require different levels of intrusiveness. For instance, the per-hop graph indicating which application components communicate with which others can be obtained by simply sniffing the network traffic [1][14]. On the other hand, derivation of flows or end-to-end dependencies requires the monitor to become application-aware [4][8], for instance by requiring the insertion of a *transaction-id* in the requests flowing through the system. This level of intrusiveness can be prohibitive in production systems.

C. Characterizing State-of-the-art

The research literature contains several techniques for debugging performance faults in data centers. These techniques require different levels of instrumentation and intrusiveness. Figure 2 classifies these approaches along these two dimensions.

Many of these techniques require information about *per-hop* resource consumption and thus require instrumentation over *all* nodes. [8] and [11] propose to use Tree Augmented Bayesian Network (TAN) [9], [10] models to identify correlations between resource usage of individual components and end-to-end SLO violation. [6] builds a decision tree in which nodes represent resources or their properties and leaves represent failed or successful requests. Nodes on the paths from root to the leaves that represent failure are diagnosed as potential root-causes of the failure. Bahl et. al. [3] propose a more intrusive technique and require information about the

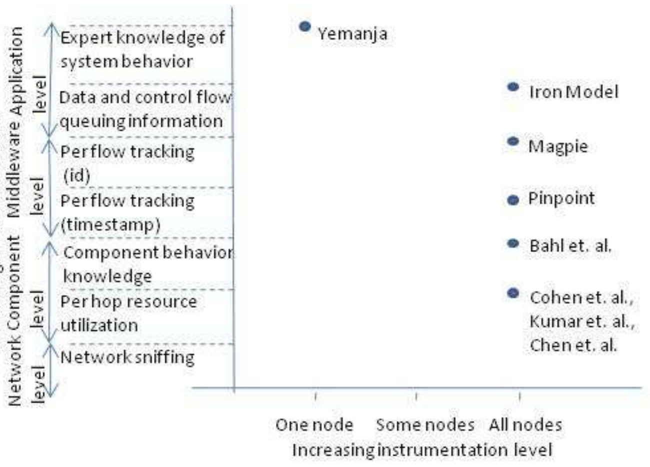


Figure 2: Instrumentation and Intrusiveness of existing performance debugging techniques

presence of load-balancers, fail-over mechanisms, etc. in the system. [3], [5] constructs a dependency graph and then use graph traversal techniques to infer root-cause nodes that best explain performance degradation. Pinpoint [7] and Magpie [4] demand information about the request-component dependencies and the request flow respectively. Collection of this information demands intrusion at the middleware or the application level. Pinpoint [7] uses data clustering techniques to discover root-cause nodes of performance degradation while Magpie [4] uses a stochastic context-free grammar (SCFG) to identify root causes of anomalies. IronModel [12] proposes an even more intrusive technique by using the per-hop data and control flow information to build a queuing model of the system. We note, however, that Magpie [4] and IronModel [12]can also be utilized for tasks beyond just performance debugging (e.g., for tasks such as capacity planning, workload analysis, and what-if analysis).The most intrusive technique is the rule-based approach proposed in Yemanja [2]. Yemanja uses expert knowledge of system behaviour to identify the root-cause(s) of the observed failures. Based on this discussion, and as visually depicted in Figure 2, we observe that:

Observation 2: Most techniques require high instrumentation or high intrusiveness, or both.

Based on the above discussions, it appears that the extent of instrumentation and amount of intrusiveness of monitors complement each other. Techniques demanding heavy instrumentation use less intrusive monitors, while techniques requiring smaller amount of instrumentation generally rely upon intrusive monitoring. For instance, Pinpoint requires request-component dependency. This information can be obtained in a high instrumentation-low intrusive manner by making each node monitor the event of request arrival. Instrumentation is required at each node but each node requires a middleware level intrusiveness to capture the information about the arriving requests. On the other hand, the request-component dependency information can also be obtained in a low instrumentation-high intrusive manner by making each request store the information of the component it passes through. Each request thus performs an intrusive operation of modifying the application content to contain the component information. The request-component dependencies can thus be obtained from the requests without instrumentation of any of the system components. We can summarize this insight as follows:

Observation 3: It is possible to tradeoff the level of instrumentation against the level of intrusiveness needed for a technique.

While existing techniques enable effective performance debugging by using either significant instrumentation or intrusiveness, production systems can place significant restrictions on which nodes can be instrumented as well as the level of intrusiveness that is permitted. This can complicate the deployment of existing performance debugging techniques in production systems or limit their effectiveness. This limitation leads us to pose our *"more with less"* question:

Is it possible to achieve effective performance debugging using low instrumentation ***and*** *low intrusiveness?*

In what follows we consider the production system depicted in Figure 1 as well as the constraints imposed in this system to propose a straw man approach for addressing the above question.

III. DOING MORE WITH LESS: AN EXAMPLE

In this section, we consider the problem of localizing violations in the end-to-end latency requirements of requests to the node (a server or a process) that is primarily responsible for the violations. For this exposition, we make the following simplifying assumption: End-to-end latency violation is caused by only one faulty node[3]. We will consider the equity trade plant application (Figure 1), along with the constraints imposed by the production environment on the permitted instrumentation and intrusiveness, for our discussion. We will discuss a straw man approach for fault localization in this environment, with the main goal of articulating the challenges in doing more with less.

A. A Production Data Center: Characteristics and Constraints

The equity trade plant, a subset of which was depicted in Figure 1, is shown in full detail in Figure 3. This environment consists of 469 nodes, 2,072 links, and 39,567 unique paths. There are 121 source nodes, 112 exit nodes, and 236 intermediate nodes in the graph. Each node in the graph represents an application component that processes the trading orders and forwards them to the downstream node or the exchange. Each server may host one or more application components. Requests enter the system from any one of the source nodes, flow through a number of intermediate nodes, and exit from any one of the exit nodes.

As indicated earlier, a critical business requirement in this environment is that the end-to-end latency for processing each equity trade should not exceed 7-10*ms*. In the event this service level objective (SLO) is violated, we need to detect, localize, and correct the *"fault"* in near-real-time. We assume that a violation of the SLO is caused either by an overload at a node or due to software or hardware errors (e.g., a memory leak) that slow down request processing. Our discussion here ignores fail-stop faults, where a component fails completely and stops processing requests; such faults are just as critical to correct but can be easier to localize than SLO violations.

This environment imposes severe restrictions on the permitted instrumentation and the intrusiveness. Due to the very stringent end-to-end latency requirements, the administrators do not allow any intermediate node to be instrumented purely for performance debugging (to prevent the instrumentation from impacting the request processing latency at the node and also to avoid any accidental changes to the system behavior caused by such instrumentation). To monitor SLA compliance, the environment timestamps each request upon arrival at an entry node and just prior to departure at an exit node. Based on these timestamps, when a request departs at an exit node, the monitor flags the request as having met or violated the SLO requirement. If the monitors at one or more exit nodes begin to observe SLO violations, our goal is to localize the fault to the node that is the root-cause of the SLO violation. Beyond the fact that certain monitors are observing violations, *no additional information is available* to us other than the system graph shown in Figure 3.

[3] We will describe the challenges in handling multiple simultaneous failures in Section IV.

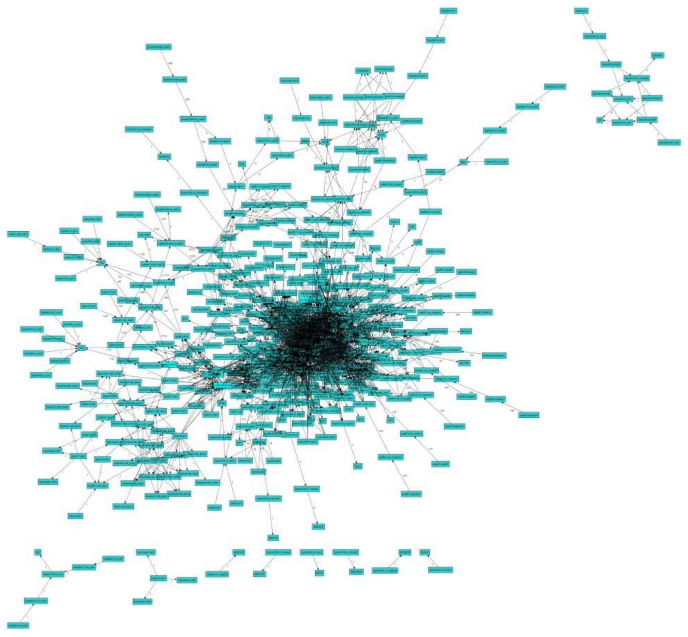


Figure 3. Real trade-plant production system graph overview

We formally state the problem as follows: Given (1) a system graph of normal operations depicting application component interactions and (2) instrumentation at the entry and exit nodes that timestamp requests; determine (localize) which node in the graph is causing SLO violations whenever one or more exit nodes observe such violations. Figure 4 depicts an example graph with such a setup. In what follows, we present two straw man approaches for addressing this problem.

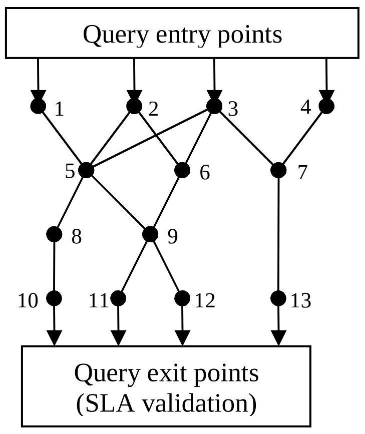


Figure 4. An example graph with monitoring at entry and exit nodes.

B. *Signature-based Localization*

The main insight behind our straw man approach is that the effect of a faulty (overloaded) node is typically visible at multiple exit nodes. This is because; once a node is the cause of a performance fault, then all requests passing through the node will experience degraded performance, and hence all exit nodes reachable from the faulty node will experience SLO violations. By identifying the exit nodes that experience SLO violations, we can localize the fault to a unique node (or a small set of nodes).

To formalize this idea, we define the notion of a *node signature.* Intuitively, the signature of a node is the set of all monitors that are reachable from this node (i.e., have a path from the node to them). Formally, for a given set of *k* monitors, a *node signature* is a *k*-bit string where each bit represents the accessibility of the monitor from the node. Each reachable monitor from a particular node has its bit set to '1' in the node signature, unreachable egress points have their bits set to '0'.

For instance, consider the graph shown in Figure 4 with nodes 10, 11, 12, and 13 as the monitor nodes. This setup generates *4*-bit node signatures. For example, signature of node 8 is 1000. Based on the graph property and the location of monitor nodes, some nodes may have the same signature. For instance, nodes 1, 2, and 5 share the same 1110 signature, nodes 4 and 7 have 0001 and, nodes 6 and 9 have 0110.

Whenever a node experiences a performance fault, one or more exit monitors will observe SLO violations within an observation window. We can compute the *violation signature* as a bit string where each monitor observing an SLO violation sets its bit to 1, while monitors that do not observe SLO violations set their bits to 0. The fault localization task then becomes the task of determining which node in the graph could have generated that violation signature. This can be determined by matching the violation signature to the node signatures; a unique match identifies the failed node precisely.

A signature match identifies the failed node because (i) only that node has a path to all the monitors that observed violations and no others, and (ii) assuming that the failed node sent at least one request along each outgoing edge, only that set of monitors would have all observed violations in that observation window.

This matching can be performed by a simple off-line approach that pre-computes the signatures of all nodes in the graph and performs a table lookup to determine which node(s) match the violation signature. The effectiveness of the approach depends critically on the assumption that most nodes will have unique signature. This depends on the graph structure and may not always be true. For example, in the graph depicted in Figure 4, nodes 1, 2, and 5 all have identical signatures of 1110. Thus, if the violation signature is 1110, this approach can narrow the failure to nodes 1, 2, and 5 but cannot pinpoint the failure to a particular node in this set. In other words, the efficacy of this approach depends on the system graph.

We have applied the signature-based approach to the production system shown in Figure 3. With the naïve approach of placing monitors at all the 112 exit nodes, we can generate a 112bit signature for the remaining 357 non-exit nodes. This naïve approach generates 137 unique signatures for the 357 non-exit nodes, which is 38% of the total number of non-exit nodes. Considering only the 121 source nodes, 71 unique signatures are generated, which is 58% of the total number of source nodes. Thus, in this production system, between 40-60% of the node signatures are unique. A failure will cause multiple node matches, allowing us to narrow the failure to a subset of the nodes and other techniques will be necessary to determine the precise cause.

C. *Online Signature Matching via Graph Coloring*

Pre-computing node signatures offline and using a table lookup for signature matching is effective only when the system graph is static. In large data centers, the graph may change over time due to application or hardware modifications (which causes nodes and links to be modified). We now present a graph coloring technique that is equivalent to the signature matching approach but is more suitable for settings where changes to the graph structure are frequent. An example of such a scenario is the dissemination of market updates in the trade plant, which is done via a publish-subscribe system. As stock traders change or add subscriptions and as new sources of market information appear, the graph topology will change.

Our graph coloring-based approach is an on-line technique that can be used to localize faults on-the-fly when an SLA violation is detected. This approach does not require pre-computation of signatures and can therefore accommodate graph changes easily.

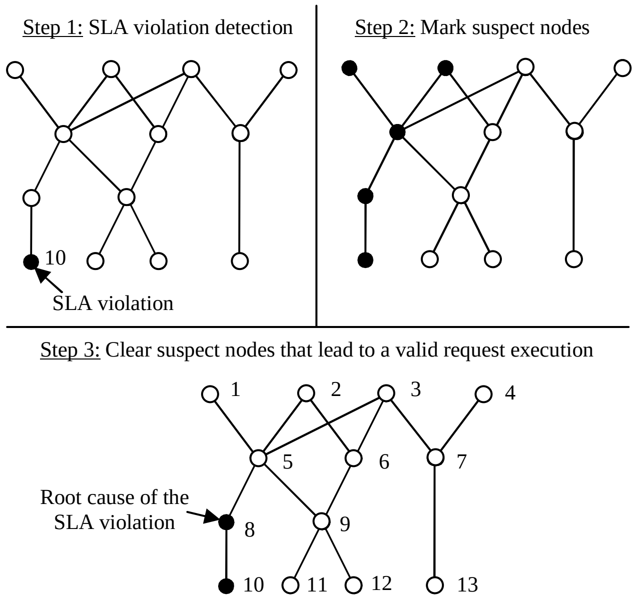

Step 1: SLA violation detection
Step 2: Mark suspect nodes
10
SLA violation
Step 3: Clear suspect nodes that lead to a valid request execution
1
2
3
4
5
6
7
Root cause of the SLA violation
8
9
10
11
12
13

Figure 5. Graph Coloring Algorithm

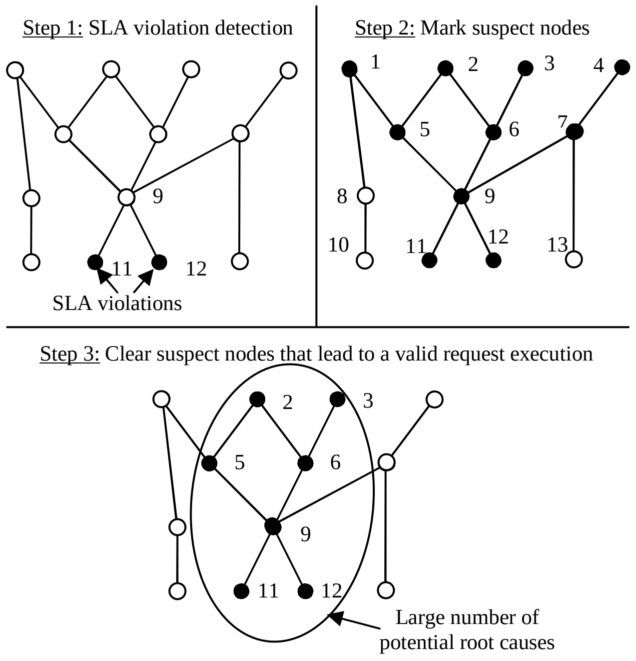
We first explain the intuition behind this approach. Consider a faulty node in the graph that causes SLO violations; assume that these SLO violations are detected at a subset of the exit monitor nodes. If a monitor observes a violation, it follows that one of its *ancestor* nodes in the graph is faulty. Since this is true for each monitor that observes a violation, and since there is only one faulty node (based on our assumption), the *intersection* of the ancestor nodes of all monitors observing SLO violations must contain the faulty node (since this faulty node causes SLO violations at all of these monitors, it must belong to the ancestors set of all of them). By computing the intersection of the ancestor set, we can narrow down the choice to a small set of nodes. Since the intersection may not yield a unique node, we must prune the graph further.

To do so, we consider monitor nodes that *did not* observe a violation. For each such monitor, it follows that all of its ancestor nodes are operating normally and did not experience a fault. Hence, we can compute the ancestor set of non-faulty nodes for each such monitor and then prune the above intersection set by dropping nodes that belong to both sets (i.e., if a node is not faulty, it is dropped from the above intersection set). By repeating this procedure for each monitor that did not observe a violation, we can pinpoint the faulty node in the system. It can be shown that this technique is functionally equivalent to signature matching---if signature match yields a unique faulty node, this technique will also pinpoint that node. If signature matching yields multiple matches, our pruning process will be left with the same set of nodes.

We can instantiate this idea as a graph coloring problem. Again we assume that if a node is faulty, at least one request was sent on each of its outgoing edge in our observation window. The algorithm can be described using the following 6 steps:

1. Mark *red* all the monitor nodes where SLO violations have been detected.
2. Mark *green* all the remaining monitor nodes where performance degradation has not been detected.
3. Mark all nodes in the predecessor graph of all the red exit nodes as red one at a time. Each time a node is marked red, increment its red count.
4. Drop all red nodes whose red count is smaller than *n*, where *n* the number of monitors observing SLO violations.
5. Mark all nodes in the predecessor graph of all the green exit nodes as green, potentially turning red nodes back to green.
6. Remaining red components are possible root causes for the observed performance degradation.

Assume that in the graph shown in Figure 5, the monitors are placed on the exit nodes, i.e., nodes 10, 11, 12, and 13. Consider a scenario where node 8 fails. As shown in Figure 5, a failure is detected only at monitor node 10 (step 1). All nodes leading to node 10 are marked red in step 2. As nodes 11, 12 and 13 did not detect an SLA violation; all nodes leading to these monitors are marked green. This only leaves node 8 as the potential root cause node for the failure.



Step 1: SLA violation detection
9
11
12
SLA violations
Step 2: Mark suspect nodes
1
2
3
4
5
6
7
8
9
10
11
12
13
Step 3: Clear suspect nodes that lead to a valid request execution
2
3
5
6
9
11
12
Large number of potential root causes

Figure 6. An instance of graph coloring with a large number of potential root causes

However, if we consider the other use case depicted in Figure 6 where node 9 fails; the failure is detected at monitor nodes 11 and 12. The algorithm outputs five possibly potential root cause nodes 2, 3, 5, 6, and 9. Like the signature matching, this is a limitation of the approach, where the graph structure prevents unique identification of the faulty nodes and yields multiple root cause nodes. We discuss further limitations and challenges in the next section.

IV. Opportunities and Challenges of Performance Debugging in Real Production Systems

The straw man approaches proposed in Section III appear promising; however, they have several limitations. In this section, we discuss these limitations and derive a set of research challenges for developing practical and effective performance debugging solutions. As discussed in Section II, operational data centers require techniques that scale with their size and complexity. Performance debugging must be achieved with minimal intrusiveness and instrumentation. As mentioned in Section I, performance debugging can be split into four tasks: build a model of normal operations of a system, deploy monitors to probe for operational statistics, detect performance failures, and localize faults. Next we discuss the challenges involved in each task.

A. Deriving a System Model

The size and complexity of real production systems are typically too large for a human to manually derive an accurate model of the system. Hence, the dependency between application components, the connection graph and flow-level information must be determined automatically. This model must be kept up-to-date with each incremental change in the hardware and software components. Thus, the objective of the modeling task is to automatically build a model of normal system behavior. Keeping the various constraints of deployment in real production systems in mind, this model needs to be built with reasonably low instrumentation and low intrusiveness. Various challenges need to be addressed while developing a system model.

As discussed in Section II, system models can be built in various ways using different system information such as request flows, per node resource utilization, etc. Techniques demanding high instrumentation and high intrusiveness can have prohibitive cost in most production systems. Further, many of these models cannot be derived in production systems because of unavailability of the required information. For instance, the production systems do not necessarily have a transaction-id that is carried throughout the system to capture end-to-end flow information of each request and must resort to other methods if such a model is necessary.

Several mechanisms can be employed to derive the system model at low cost. Low instrumentation techniques such as network packet sniffing can be employed to derive a graph of communication patterns. Such models can also be derived by examining application logs, where available, to determine which components communicate with what other components. Models depicting request flows can be derived using inference methods on application request logs of neighboring nodes—by determining correlations between certain requests at node *i* that trigger certain other request at a neighboring node *j* (instead of requiring a unique request or transaction ID that is carried with the request as it flows through the system).

In all of these cases, the completeness and accuracy of the system model depends on the accuracy of the available system information. System logs can contain erroneous entries or incomplete information, which makes it even harder to infer an accurate model of the system.

B. Monitor Placement

In order to measure the end-to-end performance metrics such as latency, throughput, etc., there is a need to place monitors in the system to gather real-time data. The deployment of monitors involves instrumentation overhead. Furthermore, depending on what data needs to be gathered by each monitor, the level of intrusiveness will vary.

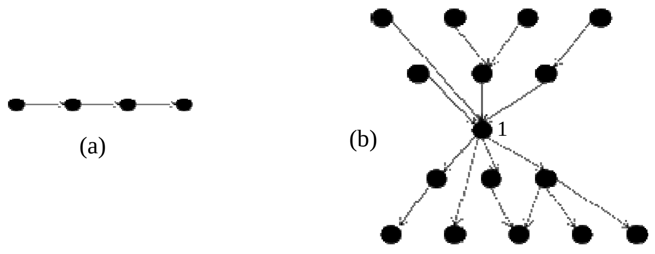


Figure 7. (a) A linear chain (b) An '*hour-glass*' graph

Since production systems are loath to incur the extra overheads of monitoring, there is a need to minimize the number of monitors placed in the system (while respecting constraints on where monitors can be placed). As argued earlier, there is often a tradeoff between instrumentation and intrusiveness—as the number of monitors is decreased, the amount of data gathered by each monitor will need to increase in order to maintain the same level of *visibility* into the data center. Thus, a key challenge is to design techniques that require low instrumentation while requiring low intrusiveness as well (in terms of data gathered by each node).

The objective of the monitor placement problem is to find the minimal number of nodes where the monitors should be placed such that the system can be monitored for detection and localization of all performance problems of interest. The problem is somewhat similar to network tomography [13], where observations at a few edge nodes are used to infer what is happening inside the network; here we use strategically placed monitors to infer the state of internal nodes in the data center.

There is also a tradeoff between the number of monitors and the accuracy of the fault detection and localization. A small number of monitors will provide less *coverage* of the data center nodes for fault detection. Similarly, in our signature-based fault localization approach, use of a smaller number of monitors increases the chance of signature collisions (and yields fewer nodes with unique signatures). Conversely, a larger number of strategically placed monitors increases the number of nodes with unique signatures and enhances the precision of fault localization. In the ideal case, *n* unique signatures can be generated with log(*n)* monitors, but the structure of the graph affects the distribution of these signatures across different nodes in the graph. For example, in case of a linear chain as shown in Figure 7 (a) with the monitor node placed on the exit node, all the nodes in the linear chain would have the same signature. Another example is of an *"hour-glass"* shaped graph as shown in Figure 7 (b) with the exit nodes as the monitor nodes. In this case, the signature of all nodes that are "*above*" the bottleneck node 1, will have the same signature.

Thus, monitor placement will need to consider the tradeoffs between the graph structure, the number of monitors, the uniqueness of signatures, and the level of intrusiveness needed to achieve a certain level of visibility into the data center state.

Constraints imposed by production systems further complicate monitor placement by eliminating certain nodes from being monitors even if they are strategically important within the system graph.

C. Real-Time Failure Detection

The failure of a node manifests itself in the form of SLO violations at monitor nodes (since the performance metric of interest exceeds the SLO threshold). A quick detection of failure can lead to timely corrective actions. The objective of failure detection is to quickly and accurately detect the presence of failures at internal nodes based on the observation being made at the monitor nodes. Failure detection becomes a challenging problem because of various factors such as the duration failure, nature of failure propagation, etc.

As argued in Section II, *point* observations at monitors can sometimes yield false positives in terms of end-to-end performance metrics. An example of such situation is when the node specific processing latency increases sharply but still does not cause the end-to-end latency to exceed the SLO threshold. Also, in situation where load-balancers shift load from one node to another, each node will observe a latency change but the end-to-end performance of the queries might stay unchanged. Clearly, failure detection should not incur false positives, or even worse, false negatives. False negatives can occur if monitor placement does not yield full coverage of nodes in the data center, and hence monitors are unable to detect failures at certain nodes.

Failure detection techniques should also be able to distinguish between effects due to valid changes in the workload and those caused by node failures. For instance, a failure of a system component and an increase in the system workload can both lead to degradation in the end-to-end performance metric observed at the monitors. Monitors will need to differentiate between these two effects.

Finally we have implicitly assumed that a faulty node impacts *all* requests flowing through it. This may not always be true in real systems. For example, queries that access very large tables on a database server may incur long processing latencies, while those accessing normal-sized tables may not see any SLO violations. In such scenarios, only a subset of the requests flowing through a node will see SLO violations. The monitor must be able to detect failures that affect only portions of the workload. The uncertainties caused by transient failures add further challenge to failure detection.

D. Fault Localization

The straw man approach proposed in Section III, while promising, has several limitations that will need to be addressed prior to deployment in a real system. We have already pointed out the limitation of the approach in localizing faults whenever multiple nodes in the graph have the same signature. Further, our discussion assumed that only one node fails at any given time. This assumption will not hold in large data centers with hundreds or thousands of servers; multiple node failures will be the common case in such systems. When multiple nodes fail simultaneously, the monitor nodes will observe a *composite* violation signature that is the union of the signature of the failed nodes. The fault localization technique will need to identify the failed nodes by "matching" the composite signature to multiple nodes that collectively generate this signature—a more challenging matching task.

Another assumption we made is that the failed node sends at least one request along each outgoing edge within an observation window. In real systems, edges are likely to be traversed with non-uniform probability. Infrequently traversed edges may not see any requests within the observation interval, resulting in a *partial* signature. A partial signature results when certain monitors that are reachable from a failed node do not see any failed request, causing their bits to be set to zero. Matching partial signature to node signatures is analogous to substring matching and can undermine the ability of the technique to localize failures to a unique node.

Finally, certain failures may be transient where the failure occurs for a short amount of time, typically when the component is close to saturation point on one of its resources. This might cause the violation signature to fluctuate with time, as the node fluctuates above or below its saturation threshold, causing some requests to fail the SLO objectives while others meet the performance objective. The inability of the technique to obtain a "fix" on the violation signature also complicates fault localization.

Each monitor must also address the inherent non-determinism in real systems. For example, load-balancing components might dynamically alter the flow of certain requests through replicated components. It is challenging to address the impact of such non-determinism on the fault propagation model. Locating faulty nodes in such context might require techniques similar to those employed in network tomography.

Although our discussion has considered the impact of real-world effects on the signature-based localization technique, the impact of multiple failures, non-uniform request flows, transient failures, and inherent non-determinism in the system will need to be considered by any type of fault localization mechanism.

V. Conclusions

With the increasing scale and complexity of information technology plants, detecting and localizing performance faults in real-time has become both a pressing need and a challenge. While several approaches for performance debugging in data centers have been proposed, these techniques do not assume any constraints on the operational data needed to detect and localize faults. We argued that collecting such operational data requires significant instrumentation or intrusiveness and that production data centers can impose significant restrictions on the degree of instrumentation and intrusiveness that is permitted in the system. Such constraints complicate the deployment of existing techniques in real-world operational systems or limit their effectiveness. Based on these insights, we indicated that performance debugging can become practical and effective in real-world systems only if they require low levels of instrumentation and intrusiveness. We then posed our "more with less questions" of whether it is possible to develop effective fault detection and localization techniques with low instrumentation and intrusiveness. We proposed a straw man approach for localizing faults by considering constraints imposed in a real system. We then presented several issues and challenges in making such approaches practical and effective in production systems.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. in Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, Bolton Landing, NY, USA.

[2] K. Appleby, G. Goldszmidt, and M. Steinder. Yemanja – A layered event correlation engine for multi-domain server farms. In IM 2001: Proceedings of the 9th IFIP/IEEE International Symposium on Integrated Network Management, pages 329–344, May 2001.

[3] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, A. D. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In Proceedings of the 2007 conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Kyoto, Japan, pages 13–24. ACM, New York, USA, Oct. 2007.

[4] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modeling and performance-aware systems. In HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems, Berkeley, CA, USA, 2003. USENIX Association.

[5] D. Breitgand, E. Henis, E. Ladan-Mozes, O. Shehory, and E. Yerushalmi. Root-cause analysis of SAN performance problems: an I/O path affine search approach. In IM 2005: 9th IFIP/IEEE International Symposium on Integrated Network Management, pages 251–264, May 2005.

[6] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure diagnosis using decision trees. In International conference on Autonomic Computing, pages 36–43, May 2004.

[7] M. Y. Chen, E. Kiciman, E. Fratkin, O. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In Proceedings of the International conference on Dependable Systems and Networks, pages 595–604, 2002.

[8] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In Proceedings of the 6th conference on Symposium on Operating Systems Design and Implementation, San Francisco, CA, USA. USENIX Association, Berkeley, CA, USA, Dec. 2004.

[9] D. Heckerman, D. Geiger, and D. Chickering. Learning bayesian networks: The combination of knowledge and statistical data. In Machine Learning, pages 197–243, 1995.

[10] D. Heckerman, D. Geiger, and D. M. Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. In Machine Learning, Springer Netherlands, Volume 20, Number 3, pages197–243, Sep. 1995.

[11] V. Kumar, S. Iyer, Y. Chen, A. Sahai, and K. Schwan. A state space approach to SLA based management. In NOMS 2008: Proceedings of the IEEE/IFIP Network Operations and Management Symposium, Salvador, Brazil, pages 192–199, Apr. 2008.

[12] E. Thereska and G. R. Ganger. Ironmodel: Robust performance models in the wild. In SIGMETRICS '08: Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pages 253–264, New York, NY, USA, 2008. ACM.

[13] T. Bu, N. Duffield, F. L. Presti, and D. Towsley. Network Tomography on General Topology. In Proceedings of ACM SIGMETRICS 2002.

[14] X. Chen, M. Zhang, Z. M. Mao, and V. Bahl, Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions. In Proceedings of OSDI 2008, San Diego, California, USA, Dec. 2008.