SYSTEMS, MODELS AND ALGORITHMS FOR FAILURE DIAGNOSIS IN NETWORKED INFRASTRUCTURE

BY

VIPUL HARSH

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

ABSTRACT

Failure incidents in networked systems such as datacenters, private or public cloud environments, WAN and enterprise networks lead to significant downtime and violations of service-level agreements, incurring large financial losses. To mitigate failures quickly, operators seek to implement automated failure diagnosis, also known as Root Cause Analysis (RCA). The goal of RCA is to localize faults with high accuracy in a timely manner using practical monitoring telemetry. There are two major challenges in designing such RCA solutions: (1) accurately modeling the behaviour of the system and (2) using the model to infer the root causes accurately. Existing works on RCA either (1) fall short in modeling the complexities of an environment and hence utilize imprecise models or (2) utilize an inference algorithm for RCA that is inadequate in extracting high accuracy from the model.

In this thesis, we propose solutions for RCA based on two high level ideas. First, we utilize a suitably designed *Probabilistic Graphical Model* (PGM) to accurately model the system behaviour. A PGM is a model based on an underlying graph structure that, as we show in the thesis, naturally allows us to model the complexities of a distributed system with many components. Second, we design custom training and inference algorithms to extract maximum inference accuracy from the PGM-based model. Our algorithms are designed to handle the various characteristics of real world systems such as large scale or noisy input monitoring data.

Using the high level ideas outlined above, we propose three RCA systems. In chapter 2, we describe Murphy, a RCA system for diagnosing performance issues in distributed applications. Murphy models weak inter-dependencies between system components via a Markov Random Field (a type of PGM based on undirected graphs) that can handle cyclic dependencies. With its more precise modeling of inter-dependencies, it is able to predict root causes with higher accuracy than past works. In chapter 3, we describe Flock that localizes unreported failures in datacenter networks. Flock models the problem via a Bayesian network (a type of PGM based on directed acyclic graphs) that can handle the various complexities of a datacenter network such as lack of knowledge of path information of flows or occasional noisy packet drops. Flock utilizes a custom inference algorithm that can speed up inference in discrete-valued PGMs by multiple orders of magnitude. This inference algorithm allows Flock to unlock the benefits of PGM-based modeling for datacenter networks and allows it to achieve higher inference accuracy than past works. In chapter 4, we describe FaultFerence, a RCA system for detecting faults in datacenter networks with only passive monitoring that obscures path information up to a large set of paths, for all flows. In such cases, even Flock (or any fault localization algorithm) can only localize the fault to a large equivalent set of devices. To localize within this equivalence set, FaultFerence uses a similar model as Flock's but additionally runs iterations of inference followed by strategically chosen "microactions" that tweak the network ever so slightly which helps in symmetry breaking. Using this approach, FaultFerence is able to reduce the time to localize faults, while also reducing the invasiveness of the localization procedure compared to ad-hoc techniques employed today. The above systems demonstrate the benefits of principled inference via accurate modeling of system behaviour towards achieving high diagnosis accuracy.

To my family

ACKNOWLEDGEMENTS

I am immensely grateful to my advisor, Brighten Godfrey for being an endless source of encouragement, creativity and enthusiasm. When I was deciding on an advisor for my PhD, someone had advised me to choose the person I would want to be like. Now in retrospect, I can say that Brighten has been a role model, both as an advisor and as a person, and I plan to emulate his many ways during my career. He has been a tireless mentor, a patient supervisor and an inspiring collaborator. I have also been repeatedly amazed by Brighten's deep insights and his remarkable technical acumen. I consider myself lucky to have had him as my advisor. I would also like to thank my committee members Sujata Banerjee, Radhika Mittal and Chandra Chekuri for their feedback which helped in shaping this thesis.

I was fortunate to be mentored by great researchers in grad school. I would like to thank Sujata Banerjee and Radhika Niranjan Mysore for taking me as an intern at VMware Research, that led to a productive collaboration. They taught me how to focus on the larger picture and how to craft a well-formulated problem. They were very patient with me, even during periods of slow progress. I thank Edgar Solomonik with whom I collaborated on two papers on parallel algorithms. Edgar's passion for research was inspiring and I will cherish our energetic discussions. I am thankful to Laxmikant Kale, my Masters advisor who was very kind and encouraging early on in grad school. His energy and enthusiasm was infectious.

I had the good fortune to collaborate with several colleagues at UIUC. I would like to thank Tong Meng, Sangeetha A. Jyothi, Sachin Ashok, Rahul Bothra, Wentao Yang, Wenxuan Zhou, Ann Zhou and Kapil Agrawal. In each collaborator I have found qualities I felt I could learn from, without exception. I only hope that along the way, I have picked up some of their skills. I also thank my labmates in the sysnet group: Sangeetha, Nathan, Tong, William, Bingzhe, Ali, KY, Aditi, Sachin, Gerard, Gangmuk, Aditya, Rahul, Ammar, Milind and other students of the larger systems group, for providing a sounding board for my ideas and for several stimulating conversations.

I deeply cherish the friendships I made at UIUC. I thank my early roommates– Umang and Vipul. They nudged me into thinking carefully about my PhD trajectory and I am grateful for their honest feedback. I fondly remember several of our conversations over chai. I thank Omkar who helped me when I was transitioning between labs. I thank Amitha, Prashant, Rachneet, Atul, Unnat and Anshika for several snacking sessions, which I could look forward to, even during the most difficult of days. I thank Samartha and Ashwini for their friendship during my initial years of PhD. Their passion was something that I always admired. I remember our collaboration on a project whose meetings would at times turn into banter sessions and laugh riots. I thank Anant, Ushashi and Brinda for several Friday night outings during the final years. These outings brought joy at times when it seemed impossible. I thank Anand, Payal, Rucha, Pooja, Bhavesh, Saboo, Prakruthi, Sameer, Anushri, Deepika, Aniruddh, Ashok, Sachin, Devi, Rahul, Vedant, Abhishek, Vaibhav, Faisal, Simone, Nitin, Bhargava, Konik, Surya, Vaishnavi, Ashish for several potlucks and get-togethers. I thank Hameed, the owner of Jerusalam restaurant for providing home style food and for his stoic yet caring personality. I would like to thank my friends from my undergrad hostel. Whenever we meet, inevitably, there's a lot of banter and warmth. I would like to especially thank my friends in the US who made me feel welcomed to couch-surf during my visits to their places: Raghav, Navin, Sushant, Sanchit, Nilesh, Sagar and Mayank. I would like to thank Chandrakant bhau for hilarious conversations and feeding me the best food back home in India.

I would like to thank the people who were influential in instilling in me a passion for mathematics and science. I would like to thank my teachers from school. A special mention to Amol who got me excited about mathematics in school. I also thank my professors and fellow students at IIT Bombay who got me interested in algorithms and discrete math, which ultimately motivated me to pursue a Ph.D..

My family has always been my anchor. No amount of space is enough to express my gratitude for having them unconditionally by my side. I am grateful for my sister, Harshita who is my closest confidante and my best friend. Her wisdom and life outlook have an uplifting effect on me and others around her. I would also like to thank her fiancé Hari and his lovely family, who have been very welcoming and warm. Hari has a friendly and sagely presence and I look forward to our association ahead. I am grateful to my wife Shreya Arya for her companionship. I have wonderful memories of our time together in the last few years– of long walks, "water-breaks" during busy work days, hikes, long drives, and a long list of other things. Her wisdom, compassionate and joyful nature has made life more enjoyable and has made me a better person. I look forward to our journey together throughout our lives. I am also very fortunate to have acquired an amazing new family in my in-laws. My parent-in-laws have been very encouraging, kind, warm and supportive. Sakshi di, Sahil bhaiya and their little one Kian have brought many a smile and I look forward to our association ahead. I thank my extended family who have shown love and care for me over the years. Finally, I am eternally grateful to my parents (Sudha Baranwal and Mrityunjoy Barnwal) for the qualities that they have inculcated in me, their boundless love and their unwavering faith and support in my pursuits. According to an ancient Indian proverb, it is very difficult to pay back one's own parents. I couldn't agree more.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Networked systems are becoming increasingly complex. These include critical infrastructures like public or private cloud environments, datacenters, WANs, enterprise networks, large distributed services among others. These systems have increasingly large scale– they can comprise millions of components which can interact with each other in complex ways.

In such large systems, inevitably infrastructure issues and failures arise. These failure incidents have a large financial cost. For instance, a recent survey [4] concluded that the hourly cost of server downtime is > \$1M for 44% of enterprises. Similarly, a Gartner study [2] estimated the hourly cost of delays, lag and downtime for the average enterprise to be about \$330K. Given this high cost of failures, its crucial for organizations to mitigate failures quickly. Often there are service level agreements in place, which require mitigation within a certain deadline [28].

When a critical infrastructure incident occurs, system operators are called upon to resolve that issue. Using the tools at hand and domain expertise, operators are required to troubleshoot various kinds of issues. These might include anomalous performance metrics seen by the client (e.g. high client latency [44, 103], high request error-rate), network issues (e.g. loss of connectivity [50], packet drops [90], high packet latency [83], low throughput), security issues (e.g. denial of service attacks [19], intrusion), hardware failures (e.g. a rack experiencing power outage, link flaps [81]), configuration issues (e.g. misconfiguration of Kubernetes auto-scalers, unintended interaction of various controllers [69]) among many others. An incident from this vast category then needs to be resolved by the operator by manually reasoning about complex interactions between components. Such manual reasoning can be very time consuming and often infeasible for a human operator.

To deal with the complexity, an important step is to make use of automated *Root Cause Analysis (RCA)* techniques. At a high level, RCA tackles the problem of finding entity(s) that may have caused a problematic symptom. This definition is deliberately non-specific, since RCA encompasses a wide range of failure types and environments. Depending on the context, a root cause entity might be a component in the system or it might refer to an action that was taken that may have caused a change. For instance, an observed problem might be high packet drops in a datacenter and the goal of RCA might be to localize the packet drops to a device using end-to-end flow metrics seen by the host. Another instance of the problem could be an anomalous performance metric that the client is experience and the goal of RCA is to find the virtual or physical component that may have caused the anomalous metric. RCA, in some form, has been deployed by virtually all the major cloud providers as it reduces the mitigation time for failures. Thus, given its practical consequences, RCA is an important goal to achieve. RCA also offers a rich space of technical challenges that need to be solved for effective diagnosis.

There have been previous works that have tackled the goal of RCA in various contexts. One line of work has built solutions using advanced monitoring telemetry. For instance, past works such as [103, 30, 60] tackle failure diagnosis in datacenter, enterprise and cloud environments, using advanced telemetry such as fine-grained packet or request metadata. However, such solutions are ineffective for the vast majority of real deployments which do not have such advanced telemetry. Designing systems for such deployments based on commonly available telemetry is thus an important research goal for increased applicability.

In contrast, designing effective tools for common environments is hard due to myriad complexities of today's networked systems. We describe some of these complexities below-

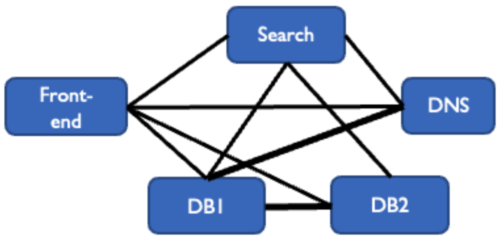
- A large distributed service might comprise thousands of "micro-services" (see figure 1.1a). To answer a single request in such a large service, thousands of micro-services would interact with each other to produce a response. Due to this reason, if a client request results in an error or experience high latency, it might be difficult to locate the exact source of the error [44].

- In a distributed service, a practical telemetry might only provide information about which pairs of services are talking to each other, but not the precise order of the call graph (see figure 1.1b). The precise order allows an operator to reason about the journey of a request as it flows through the distributed service. Previous works like [44, 103, 25] rely on the call graph of the application for root cause analysis. Not knowing the precise call graph can make troubleshooting performance issues much harder and render these works ineffective.

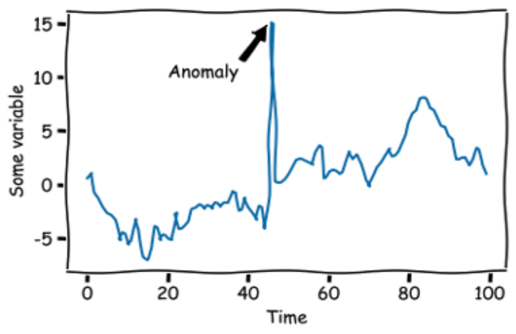
- There might be cyclic dependencies between the system components which makes root cause analysis tricky (see figure 1.1e). Without cyclic dependencies, one approach to model the system is via a "causal DAG" of components, which detail the exact set of components that any component can directly influence. The presence of cyclic dependency makes techniques [25, 44] that assume a causal DAG inapplicable.

- Usually, datacenter networks have multiple routes for any given flow and the network switches pick one of multiple next-hops using ECMP routing (via hashing on the 5-tuple of the flow: <src ip, dst, ip, src port, dst port, protocol>). The ECMP hashing takes place in real-time and in hardware for speed. Usually, it is hard to obtain knowledge of the path onto which a particular flow was hashed. Then during troubleshooting a

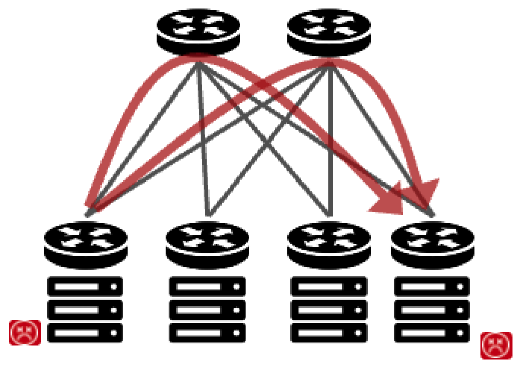
(a) Scale



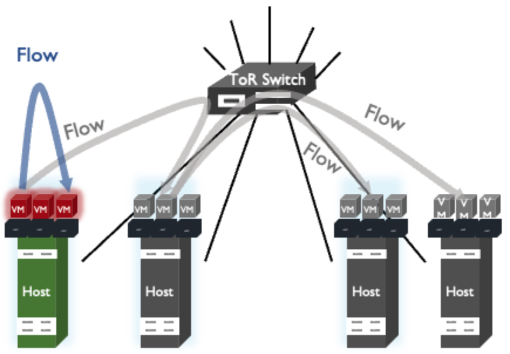
(b) Unavailable call graph



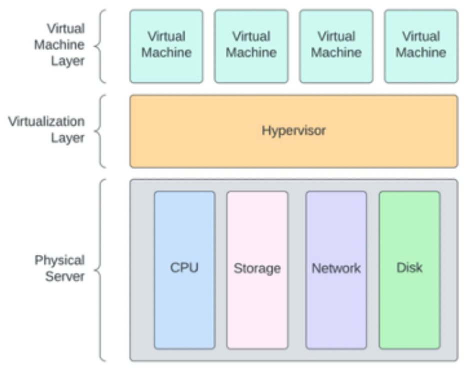
(c) Coarse-grained information



(d) Path uncertainty



(e) Cyclic Dependencies



(f) Virtualization

Figure 1.1: Examples of types of complexities present in various networked systems: (a) large scale of systems comprising hundreds of thousands of components, (b) the precise order of invocation of distributed services is not known, only the pairs of services talking to each other are known, (c) presence of coarse-grained information, aggregated over a time interval instead of fine-grained events, (d) unknown path information for a flow in a datacenter due to ECMP routing, (e) presence of cyclic dependencies between components, (f) multiple layers in a system due to virtualization, any of the virtual components can be responsible for an incident

network problem, one has to deal with "path uncertainty" (see figure 1.1d), where the route for a flow is obscured to a set of paths.

- Each component in a system executes several actions (e.g. send a request/packet, create a process etc.). In a networked system so large, its not feasible to record every action in the system. Hence, practical telemetry often consists of coarse-grained metrics that are aggregated over a period of time, usually a few minutes (e.g. number of requests sent by a microservice every 30 minutes, see figure 1.1c). Aggregation however makes it difficult to reason about the exact set of events that may have led to a problem. "Provenance" based systems that reason about events and their causality [31, 95, 53, 94, 86] don't apply in such practical settings.

- Virtualization of hardware is commonplace in networked systems. Some examples of virtualization include VMs on hosts, virtual NICs on physical NICs, virtual disk on physical storage etc. While there are several benefits of virtualization such as efficiency, cost savings, security etc., it also makes troubleshooting more complex since now virtual components may also contribute to the problem (see figure 1.1f). A virtual component (e.g. a virtual disk) can be separated in space from another component (e.g. a VM) that's interfacing with it in a way that the virtual component appears logically co-located [98]. This kind of virtualization introduces multiple extra layers and a fault may arise in any one of the underlying layers. Indeed, sometimes it may be hard to attribute a failure to a physical or a virtual component.

We note that this is not an exhaustive list. Any one or multiple of these complexities can be present in a practical deployment. Given all this complexity, one might wonder if its possible to design a RCA tool that can achieve (a) *high accuracy*, (b) is based on *practical telemetry*, and (c) has a reasonable *inference time*. To achieve high accuracy, we seek solutions that are principled and are robust to the system complexities, some of which were described above. To maintain practicality of our solutions, we want RCA techniques based on monitoring data that can be easily obtained in real deployments. Finally, to ensure that the RCA systems are useful to system operators, we want to ensure that the data fetching and the RCA algorithm finish in a few minutes, which is an acceptable amount of delay for failure diagnosis. A longer runtime (e.g. an hour) would have resulted in significant amount of downtime already and hence such solutions would be too slow.

To achieve the above goals, a RCA system needs to meet two objectives: (1) designing a model of the system using available monitoring data, capturing system behaviour as accurately as possible and (2) designing model training and inference algorithms, handling the practical challenges of the environment. Many existing RCA solutions based on commonly available monitoring data have been proposed. We can categorize existing work into two groups– (1) solutions whose model of the system is too simplistic or does not accurately capture the behaviour of the components. For instance, 007 [24] effectively models the problem in a way that does not account for traffic skew in the network. NetMedic's design [63] involves heuristics and a fixed model (as opposed to a learning-based approach), thus making implicit assumptions about the real world that may not hold (for e.g. the nature of dependencies of the same two entities can change over time; a model that gets trained continually on recent history can capture such a dependency but a fixed model can't). ExplainIT [61]'s model effectively ignores how components are organized in the system. As we will also show later, not modeling system behaviour accurately comes at the cost of reduced inference accuracy. (2) The second category of solutions include those which model the system accurately but their training or inference algorithms do not handle all practical challenges of real world systems. For e.g. Sherlock [25] models components and their interactions accurately, but its inference algorithm does not scale to large problems making such solutions inapplicable for large environments such as a datacenter network. NetBouncer's [90] implicit model is also close to the actual behaviour of the system but its inference technique does not result in high inference accuracy. Its inference is based on predicting drop rates on every link using an optimization problem. While this approach also works in some cases, as we show, it is outshined by approaches based on Bayesian inference.

To tackle the limitations of past works, we use two high level technical ideas– (1) we model a particular system via *probabilistic graphical models*(PGMs), adapting the PGM to the particular RCA problem at hand and staying as close to the actual behaviour as possible. PGMs (§ 1.1.2) are models based on an underlying graph structure and is a natural fit for the distributed nature of the systems we are trying to troubleshoot. By suitably adapting PGMs, one can capture the various complexities of real environments such as path uncertainty, cyclic dependencies between entities, noisy packet drops etc.. When possible, we learn parameters of the PGM-model from historical data. (2) Secondly, we develop training and inference algorithms that are designed with the goal of extracting maximum accuracy from the model, while solving the practical challenges of the environment such as scale of the networked system or noise in the monitoring data. Embodying these ideas, this thesis proposes three systems for root cause analysis.

We propose Murphy (§ 2), a root cause analysis system, to diagnose performance issues in distributed cloud applications. Such applications have complex inter-dependencies between distributed application components as well as network infrastructure components, making it difficult to reason about their performance. An important system behaviour in these systems is that components interact with each other in such a way that they influence each other's metrics, inducing cyclic dependencies between entities [1]. Murphy models this behaviour via a *Markov Random Field (MRF)-based model* (a type of PGM based on an underlying undirected graph structure, see § 1.1.2.1), which is learnt from historical data comprising coarse-grained timeseries metrics of entities. Murphy's MRF model is based on an undirected graph that can tolerate cycles, unlike solutions like Sage [44] that require acyclic causal DAGs between entities or ExplainIT [61] that ignores inter-dependencies altogether. To predict root cause in this model, we develop a counterfactual reasoning algorithm that predicts the effect of changing the metrics of one entity on another. This counterfactual reasoning algorithm uses the underlying MRF model to propagate the effect of a hypothetical change in a metric (e.g. CPU usage) of a candidate root cause entity to the entity that's experiencing a performance issue. To avoid noisy inference, Murphy's inference algorithm evaluates the metrics of only a small number of other entities that may get affected by the candidate root cause entity and may also be relevant to the problem. Faithfully capturing the entity inter-dependencies allows Murphy to achieve a higher inference accuracy than past works. We evaluate Murphy in an emulated microservice environment and in real incidents from a large enterprise. Compared to past work, Murphy is able to reduce diagnosis error by $\approx 1.35\times$ in restrictive environments supported by past work, and by $\geq 4.7\times$ in more general environments.

We propose Flock (§ 3) to detect failed components in a large datacenter, using end-to-end flow information observed by the end-hosts. Unlike Murphy, the challenge here is to infer hidden metrics, not present in the monitoring data e.g. unaccounted packet drops on a link. To do so, Flock models a datacenter network via a *Bayesian network* (a type of PGM based on an underlying directed acyclic graph, see § 1.1.2.2), designed specifically to capture faults in components in a network topology. This model accurately captures the complexities of a datacenter deployment such as the lack of knowledge of paths taken by flows, occasional noisy packet drops or delayed flows not indicating an error. To achieve high inference accuracy using this model, Flock uses *Maximum Likelihood Inference (MLE)*. However, as Sherlock [25] showed, existing MLE inference algorithms for Bayesian networks are intractable for size of a datacenter that comprises hundreds of thousands of components. To handle the problem of scale, we design algorithmic techniques (greedy inference and Joint Likelihood Exploration (JLE)) to accelerate inference in discrete-valued PGMs. Using faults created in simulation and a testbed, we show that Flock improves accuracy over the best previous datacenter fault localization approaches by $1.19-16\times$ on the same input telemetry,

[1]We use entities and components inter-changeably

by $1.2 - 55\times$ after incorporating passive application telemetry and by $1.66 - 12\times$ with in-network telemetry (INT) that also captures the path of flows. Flock's inference runtime is $> 10^4\times$ faster compared to past PGM-based solutions, allowing it to unlock benefits of PGM-based modeling at the scale of a datacenter. We also prove that Flock's inference algorithm correctly recovers the set of truly failed links, in restricted settings.

We propose FaultFerence (§ 4), a system to diagnose unreported failures in datacenter networks (e.g. black holes) using the most practical telemetry– passive monitoring of flows. Passive monitoring obscures the path taken by flows up to a large set of possible paths. In such cases, Flock (or any fault localization algorithm) are fundamentally limited and can only narrow down the fault to a relatively large set of equivalent components. Devices in these equivalent sets perform the same function in the network and hence appear symmetric in the monitoring telemetry. FaultFerence performs iterations of inference followed by "microactions" that very slightly tweak the state of the network. FaultFerence's inference is based on a Bayesian network-based model. This model is similar to Flock's and is able to capture system complexities such as the path uncertainty about flows. To perform localization, FaultFerence utilizes a greedy algorithm to generate a short sequence of microactions to be applied to the network. Each of these microactions helps to break symmetry of devices in an equivalence set. The coupling of FaultFerence's inference with intelligently chosen microactions allows FaultFerence to quickly localize a failure, reducing the time and invasiveness of current ad-hoc procedures. In our experiments, we localize black holes in a datacenter topology via micro-actions that alter some paths for a small set of flows. We show that FaultFerence localizes the fault $> 98\%$ of the times, and reduces the number of steps by $2.1\times$ compared to what a smart operator might manually take. We believe FaultFerence is the first failure diagnosis system to tackle the problem of localizing faults among equivalent devices, and it opens a new paradigm of closed loop root cause analysis in a system with many components.

Flock appeared in ACM CoNext, December 2023 [55]. Murphy appeared in ACM SIGCOMM, September 2023 [56] and FaultFerence's publication is forthcoming.

1.1 PRELIMINARIES

We define some terminologies that will be used throughout this thesis.

1.1.1 Failure diagnosis in networked systems

1.1.1.1 Gray failures

A gray failure is a failed component which is experiencing performance degradation or a failure, but does not get reported by existing monitoring systems [59]. An example of a gray failure is a link that is silently dropping a packet without updating switch counters or a silent packet corruption in a switch.

1.1.1.2 Microservices vs Monolith applications

Modern applications are moving from monolithic codebases to highly modular application comprising tens to thousands of microservices that perform relatively simple functions. The modular design allows each microservice to be provisioned and maintained independently. While microservice-based applications are more agile and scalable, they are also more complex to troubleshoot . For example, if the application user experiences a high delay, the reason could be in any of the several component microservices that coordinate to generate the user response (see [44] for a discussion of these issues).

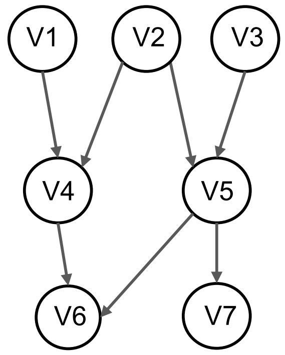
1.1.2 Brief overview of Probabilistic Graphical Models (PGMs)

A probabilistic graphical model is a probability model based on a graph where each vertex v of the model represents a random variable V. We describe two categories of PGMs (a) Markov Random Fields, based on undirected graphs or general directed graphs and (b) Bayesian Networks, based on directed acyclic graphs (DAGs). The variables in a PGM, in our context, will represent states in the system, either observed or unobserved. The PGM will define how these states depend on other states and components. For a specific RCA problem, we will make use of a model that appropriately captures the inter-dependencies between components in that environment.

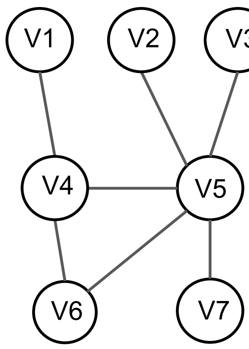
1.1.2.1 Markov Random Field

A Markov Random Field is a type of PGM where the underlying graph structure is an undirected graph. The joint probability of all vertex random variables is given by,

$$P[v_1, v_2 ... v_n] = \frac{1}{Z} \prod_{C\text{:maximal cliques}} \phi_C(V_1^C, V_2^C ...) \tag{1.1}$$



(a) A 3-layered Bayesian network (directed DAG-based PGM)



(b) A Markov Random Field (undirected PGM)

Figure 1.2: **Two types of Probabilistic Graphical Models (PGMs) depending on the underlying graph structure.** In the context of failure diagnosis, the vertices of the graph would be the entities– flows, links, VMs, switches etc. and a probabilistic model on top of the graph would describe how these entities affect each other.

Where Z is an (unknown) normalization constant and C is any maximal clique in the graph G. In the above case, the underlying graph is undirected. Such PGMs, based on undirected graphs, are known as *Markov Random Fields* (MRFs). Figure 1.2b shows a toy Markov Random Field with 7 random variables.

1.1.2.2 Bayesian/Belief Network

A *Bayesian network* is a PGM-based on a "causal" DAG, G, where the joint probability is given by,

$$P[v_1, v_2 ... v_n] = \prod_{V:\text{vertices}} P[V = v|\text{parents}(V)] \tag{1.2}$$

A Bayesian network is a generative model which captures exact directed dependencies as to which variables are caused by which other variables. For instance, in the Bayesian network shown in Figure 1.2a, random variable V4 depends only on V1 and V2, and is conditionally independent of all "non-descendent" variables- V3, V5, V7 given V1, V2. A Bayesian network can capture "causality" between the variables. For instance, the Bayesian network in Figure 1.2a shows that V6 is caused by V4 and V5. In turn, V4 is caused by V1 and V2, and V5 is caused by V2 and V3.

CHAPTER 2: MURPHY: PERFORMANCE DIAGNOSIS OF DISTRIBUTED CLOUD APPLICATIONS

2.1 INTRODUCTION

Troubleshooting IT incidents, such as slow responsiveness of a service or loss of connectivity, is getting harder due to the large number of application and infrastructure components and complexity of dependencies between them. Modern microservices-based architectures contribute to this complexity as do the increasingly disaggregated, virtualized, and distributed deployments, even spanning multiple clouds. As a result, when incidents happen, multiple teams, responsible for different infrastructure components, scramble to pinpoint the source of the outage or performance degradation.

This paper focuses on an important step of resolving incidents – *performance diagnosis*. Operators are commonly presented with an observed entity E_o (say, a backend database server) which has a problematic metric M_o (e.g., high memory usage). The goal is to find a "root cause": an entity E_r and an associated metric or property M_r which led to the observed problem (or a short list of likely root causes). Unlike fault localization works [24, 25, 90, 13, 57, 47] that infer some hidden state of components like links silently dropping packets, the challenge here is not necessarily to infer hidden metrics; indeed, performance diagnosis systems typically leverage extensive telemetry so that relevant entities (applications, containers, VMs, routers, data flows, etc.) and their associated metrics (API latency, CPU utilization, flow throughput, etc.) may be well known. Instead, the core challenge is to infer the causality relationship $(E_r, M_r) \rightsquigarrow (E_o, M_o)$.

There has been an active push in industry and in academia towards performance diagnosis – spanning from early work focused on enterprise networks [25, 63], to applications in data centers [61], to recently-renewed interest in performance diagnosis for microservices [44]. We found to our surprise, however, that these designs are limited in either accuracy or applicability of their model to our target environment. A key design choice is the kind of input they utilize to model dependencies (i.e., potential functional influence) between entities in the system, which we can classify into three types.

1. *No dependency knowledge:* ExplainIt [61] requires no knowledge of the dependency structure, but we found this results in low accuracy.

2. *Directed acyclic graph (DAG) of dependencies:* Sage [44] requires a known dependency DAG, specifically the microservice call graph. Unfortunately, real-world enterprise environments contain many cyclic dependencies. Even in the restricted setting of microser-

vices, cycles exist in practice among services within the execution of a single request [85]. Furthermore, independent requests affect each other indirectly due to resource utilization, and infrastructure components like CPUs, NICs, virtual routers, etc. introduce further bidirectional (and thus cyclic) influence. Furthermore, the specific direction of a dependency between entities can sometimes be hard to determine.

3. *Relationship graph:* Another option is to model known potential dependencies, but such that there may be cycles and the relationships themselves are loosely defined, in contrast to causal dependencies. We refer to this as a *relationship graph.* NetMedic [63] takes this approach, and is perhaps closest to meeting our needs, but we found it too resulted in poor accuracy. This may be because its inference algorithm uses fixed heuristics (as opposed to a learning-based approach) that are unable to capture complex and variable patterns in our environments.

In this paper we seek to design a performance diagnosis scheme which (a) is applicable to common enterprise environments, including cloud applications and enterprise infrastructure, while (b) achieving high diagnosis accuracy. Intuitively, this involves a choice of input information, and algorithms to effectively reason about that information.

First, Murphy is built to use telemetry from common enterprise monitoring software. Monitoring platforms, including the one we will use to test Murphy, can see entities in the system like VMs, containers, hosts, routers, TCP flows, etc., as well as relationships between them – for example, VM v_1 is located on host h_5 and it has a TCP connection to v_2. Such relationships imply a likely influence between entities, and we want to make use of this commonly-available information to improve accuracy. But the directionality of that influence might be either or both, in a way that might be dependent on the application, API call, scenario, and moment in time, and it might even be a weak, non-consequential influence. Therefore, Murphy models entity dependencies with a graphical model that can accommodate cycles, including bidirectional edges that avoid assumptions on the specific nature of the relationship between two entities; cycles thus may be the common case in the input. This input is of type (3) (a relationship graph), not unlike NetMedic.

Second, to achieve high accuracy, we need a powerful reasoning algorithm to predict causality in the relationship graph. We therefore design a new learning-based method: Murphy uses a *Markov random field (MRF)* [68][2], a type of probabilistic graphical model that can represent nodes that might simultaneously affect and be affected by their neighbors. We found it was important to train the model on demand, and developed an adaptation of

[2]MRFs have been previously used for medical diagnosis [100], pattern recognition [27], image analysis [68] among several other applications.

Gibbs sampling for inference. Using this design, Murphy learns a joint distribution of all entity metrics from historical values, which it uses to predict the impact of a potential root cause. Finally, after generating the root causes, Murphy goes one step further and generates simple human-interpretable explanations about the root causes using a threshold-based labeling scheme.

In addition to the design of Murphy, this paper describes the following evaluation results:

- **Metric prediction model selection:** Accurately predicting the effect of changing a metric is a key sub-component for performance diagnosis (and may also be of independent use). Using telemetry data from a large enterprise with over 300 production applications comprising $\approx$ 17,000 entities, we evaluate four candidate designs for this sub-task and further refine our scheme. We also show that a prediction technique which better captures cyclic influence improves accuracy, suggesting that these complex interactions are indeed present in deployed applications.

- **Diagnosis accuracy:** We evaluate Murphy and several recent schemes in two common environments, which may include cyclic inputs. **(a)** In the DeathStarBench [45] microservice benchmarking suite, we test an environment with a microservice application and its associated infrastructure which may induce cyclic relationships. Murphy achieved 86% accuracy in diagnosing injected faults, whereas NetMedic and ExplainIT achieved very low accuracy (§ 2.6.1). When we ignore cycles so that it is possible to run Sage on the input, it did not produce the root cause as it wasn't captured by its modeling. **(b)** We use an evaluation on real IT incidents from a large enterprise. While the incident set is relatively small, the result is promising: Murphy produces $\geq 4.7\times$ fewer false positives than ExplainIt and NetMedic, while producing a similar number of false negatives, when taking the operators' manual judgements as the ground truth. (Sage is not included because it cannot model this environment.)

- **Diagnosis accuracy in DAG environments:** Although handling more general non-DAG environments is our goal, Murphy should also perform well when a DAG is known and is an appropriate model of the environment. We compared accuracy using the DeathStarBench environments for which Sage was designed (§ 2.6.3). Here, Sage performs well, averaging 77% accuracy, but Murphy performs even better with 83% accuracy, illustrating the power of MRF-based reasoning. Both NetMedic and ExplainIT perform poorly in this environment.

- **Robustness**: Although using dependency information from telemetry is useful, there is a risk that the telemetry is incomplete or has errors, as is often the case with monitoring data from a large infrastructure. We evaluate robustness by introducing a series of types of input data degradation. Overall, Murphy achieves 1.5× less diagnosis error than Sage, while NetMedic and ExplainIT perform much worse.

Traces we generated from the DeathStarBench [45] benchmarking suite are available at [7]. Some of the ideas in this paper are being adopted in VMware Aria Operations for Networks [9], a commercial multi-cloud network observability[3] platform, to provide insights about infrastructure dependencies and problems. This work does not raise any ethical issues.

2.2 BACKGROUND

2.2.1 Enterprise network monitoring

Our target domain is cloud infrastructure for medium to large enterprises. A typical enterprise uses private clouds (i.e., on-premises data centers), as well as virtual infrastructure in public clouds. Workloads include both monolithic and microservice-based applications, on virtual machines (VMs) and containers. Environments continuously change as services and infrastructure components are brought up, scaled up or down, moved, and decommissioned. The scale of infrastructure varies, with hundreds up to several thousands of applications in a very large enterprise.

Most enterprise IT teams use multiple monitoring tools to gain visibility into the environment, including hosts, applications, logs, and network infrastructure. Microservice-based applications often utilize additional specialized monitoring with tracing tools like Jaeger [6] and Zipkin [12] tracking application-level metrics such as RPC latency and errors.

We describe VMware Aria Operations for Networks [9], an application-aware multi-cloud network observability platform, which is our source of experimental data in this paper and is reasonably representative of common enterprise monitoring software. This platform obtains passive telemetry from multiple data sources, including: VM management platforms and SDDC controllers (which in turn monitor individual hosts and virtual networks), physical network devices (routers, switches, firewalls, etc.), APIs from public cloud providers, and Netflow/IPFIX sources for information about data flows. This telemetry includes metadata about a variety of entity types, with generally multiple performance metrics for each entity. Each metric is a time series, collected in intervals within minutes. Data older than a day is aggregated for most metrics into longer time intervals, and is stored for a rolling summarized

[3]We use the terms observability and monitoring interchangeably.

Entity type	Example metrics
VM	CPU utilization, memory utilization, network transmit rate, receive rate, packet drops, disk read/write rate
Host	*Metrics similar to VM metrics*
Container	*Metrics similar to VM metrics*
Virtual NIC	Transmit rate, receive rate, dropped packets
Physical NIC	Transmit rate, receive rate, dropped packets, latency, interface peak buffer utilization
Flow	Session count, throughput, RTT, packet loss, retransmission ratio
Switch interface	Network rate, dropped packets, latency, interface peak buffer utilization
Datastore	Space utilization

Table 2.1: Entity types and associate metric values used in this paper.

window. Example entity types and associated metric values relevant to the present paper are listed in Table 2.1.

The monitoring platform also provides entity metadata, encoding entity associations. For instance, a VM is related to its physical host, NIC, and flows that originate or terminate at it. Flows identified by 4-tuple (source IP, destination IP, destination port, and protocol) are related to their source and destination. Metadata can also encode application definitions: Operators can tag or classify VMs comprising an application and also define “tiers” within an application (e.g., web tier, database tier, etc.). Application definition can be manual, or automatic based on tags, naming conventions, and analysis of flow communication patterns [10]. The data set used in this paper has over 300 defined applications (§ 2.5).

Network monitoring platforms can have visibility into a single data center or an entire enterprise (depending on deployment). As such, the scale can be large.[4]

Enterprise operators use monitoring tools like the above to help gain visibility into the state of the network and applications. Some platforms also detect anomalies. However, automated diagnosis is still lacking and operators rely on manual intervention, trial and error and domain knowledge of engineers across several teams to diagnose performance issues. Often the root cause is unknown even after days of manual diagnosis and disappears after operators restart some components in an attempt to remove the offending triggers.

[4]Large monitoring deployments can have hundreds of thousands of VMs monitored by a single instance of an observability platform, and larger enterprises can deploy multiple instances with federation visibility.

2.2.2 The need for handling cyclic dependencies

A key consequence of relying on commonly-available monitoring data in typical enterprise environments is that we must work with *highly complex, often cyclic, and often uncertain, dependencies.* In contrast, prior works [44, 25] expect, as input, clean causal dependencies between entities, like dependencies of the form (A–>B), where A depends on B but not vice versa. They also expect the totality of dependencies to form a causal directed acyclic graph (DAG).

Cyclic dependencies, in a system representation, can arise in two ways. The first is actual cyclic influences in the system. Even in the restricted setting of microservices, cycles exist in practice among services within the execution of a single request [85].[5] Different user-facing services can also affect each other indirectly as they can share resources and common downstream microservices. Even more cycles appear when we include infrastructure entities. Consider an example: a VM v_1 resides on host h (a similar relation exists in the incident in Figure 2.1). The CPU usage of v_1 influences the overall CPU usage of h, but because the physical CPU is a shared limited resource, this will also influence the CPU usage of another VM v_2 on h – and symmetrically, CPU usage of v_2 will affect h and v_1's CPU usage, thus forming a cyclic influence. A similar effect would occur with Kubernetes pods in burstable or best-effort mode on a shared node, as well as other shared resources like memory, disk, NIC drop rates and throughput, and indirect influence through TCP flows. All of these dependencies are present in our enterprise environment.

The second source of cyclic dependencies is when the direction or existence of dependency between two entities is unclear, and thus *the relationship graph over-approximates the actual influence.* For example, suppose there is a TCP flow from a web front end VM v_1 to another VM v_2. This flow will be visible in the monitoring system, but its exact nature is not. It might be that the performance of the application at v_1 depends on the RTT or throughput of the flow, which in turn might depend on CPU resources in a back-end database running at v_2, so that v_1 indirectly depends on v_2. Resources at v_2 may also depend on v_1, as handling the requests in the flow involves added work. Both of these patterns exist in the incident shown in Figure 2.1. But either of these possible influences might turn out to be negligible, if, for instance, the flow was logging information in the background and not involved in the web front end's primary traffic. These and other possibilities cannot be discerned only through metadata. Past works such as Orion [32] try to automatically infer the direction of influence, but require fine grained ($\approx$ 10 ms) timing information of every single request.

[5] In fact, [85] identified lack of cycles as an unrealistic aspect of current microservices testbeds.

This is out of scope for our enterprise environment.[6] Hence we add an association in both directions as an over-approximation of actual influence, creating many additional cycles and leaving it to the diagnosis algorithm to reason about actual influence in a particular incident.

The above effects combine so that cycles are the norm. Across our data set of 13 incidents in our enterprise environment, on average, the relationship graph had over 2000 cycles of length 2 and over 4000 cycles of length 3, and *all* VMs of affected applications in every incident were involved in at least one cycle.

2.2.3 Related work

There is a broad set of work that attempts to automate various aspects of troubleshooting, such as routing incoming tickets to the right team [46], orchestrating active probes to check availability or latency [50], localizing faults [24, 25, 90, 13, 57, 47] like links silently dropping packets and also performance diagnosis [63, 44, 61]. It is helpful to understand work in two main subareas.

Fault localization. A rich body of work [25, 97, 62, 74, 32, 90, 24, 66] performs fault localization, distinct from ours, in which components have well known quantifiable failure signatures. For example, Sherlock [25] models entities as being in one of three states (up, troubled, or down), and other works focus on even more specialized faults than Sherlock, e.g., physical network packet drops or link loss or delay [97, 90, 24, 66]. In contrast, our work is intended for performance diagnosis, where there is not a common definition of failure signature and diagnosis seeks to answer richer metric-related questions of the form "which component's performance metrics influence the observation the most?"

We considered trying to repurpose the underlying models of the above work for our goals, but their models do not meet the needs of our environment. [25, 32] rely on detailed models of how an entity depends on another, obtained through fine-grained (e.g., 10 ms granularity) packet timing which is not available in general enterprise environments. Also, as noted in [63], Bayesian techniques [25, 97, 62, 74] can only model acyclic dependencies; the cyclic case is fundamentally more challenging due to the resulting complex interactions among entities. We have to work with entity topologies that contain many cycles, due to lack of detailed dependency knowledge as well as true cyclic influence.

Performance diagnosis. Performance diagnosis tools solve a problem that is closest to ours. We therefore discuss them in more detail.

ExplainIt [61] performs pairwise correlations between metrics of the observed problem and

[6]If more detailed, accurate telemetry were available – for example, via eBPF – Murphy could use it. However, true cyclic dependencies will still be present.

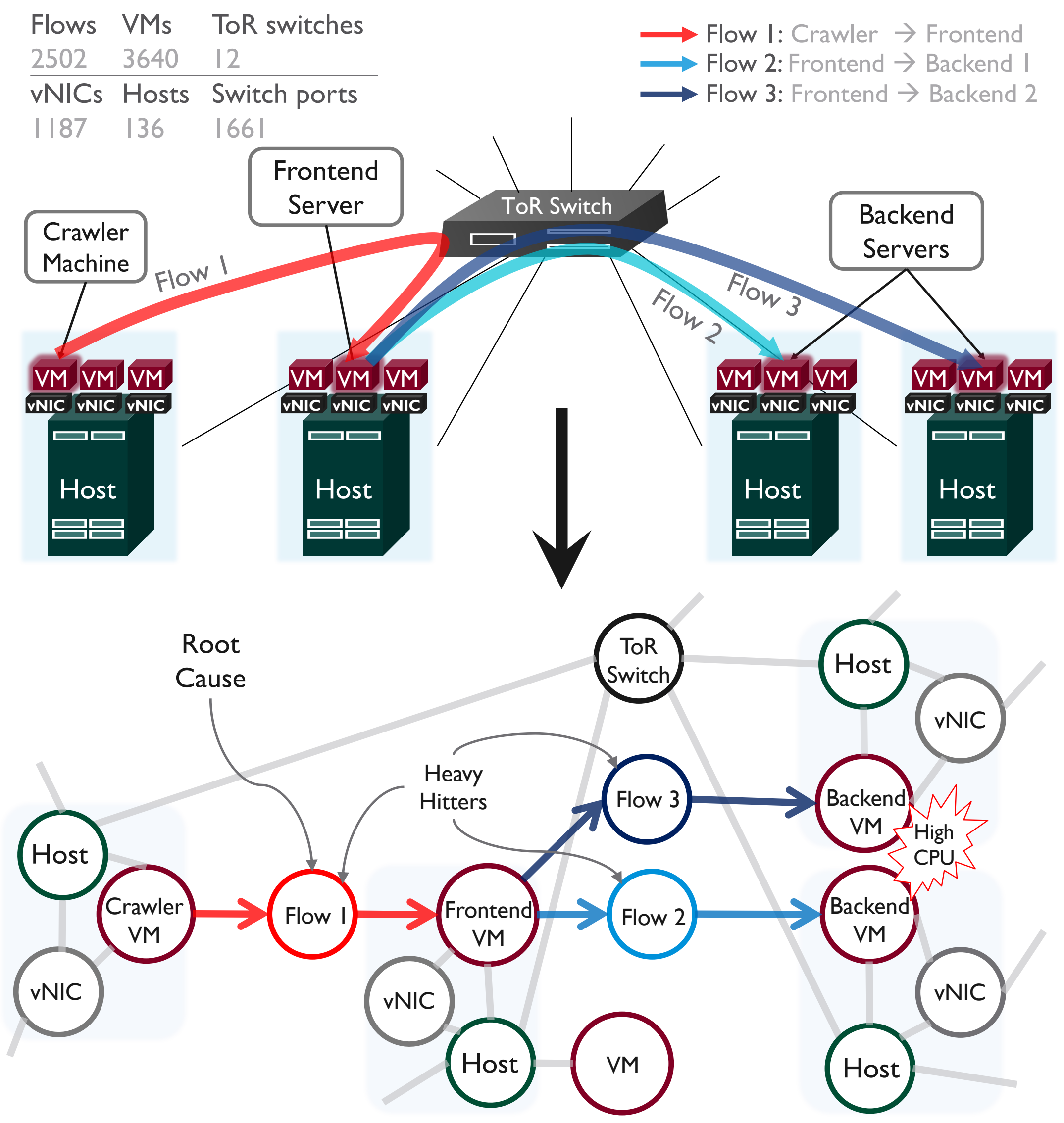


Figure 2.1: Production incident. Shown at the top is a physical topology associated with an incident where a crawler VM was sending too many queries to the frontend at a high rate. This resulted in the front-end initiating a large number of requests to the backend, causing high CPU load and application unresponsiveness. The bottom figure shows the relationship graph corresponding to the physical topology that Murphy built to analyze this incident. Only a subset of the entities are shown but the top left table enumerates the total no. of entities in our relationship graph. On analysis, Murphy correctly ascertained flow 1 as the root cause for the high CPU load at the backend and flagged it for the operator to handle.

of each candidate root cause.[7] However, as we will show, simple metric correlation-based analysis can yield inaccurate results as it does not take into account the topological structure between entities.

NetMedic [63] uses a dependency graph to capture the known entity dependency structure. It labels edges with weights based on pairwise correlation between neighbors using historical metric values, augmented with heuristics to reduce weights when metric values are roughly normal, and remove or coalesce redundant or aggregate metrics. Finally, it ranks root causes based on a geometric-mean of path weights, and a score of the global downstream impact of the candidate root cause. This approach can have a similar limitation as ExplainIt due to use of pairwise correlations. In general, NetMedic's inference is based on fixed heuristic rules (ignoring "normal" influence, geometric-mean weighting, etc.) which can be brittle in real environments. In our tests, NetMedic's accuracy was low, except with lenient definitions of the root cause (§ 2.6). This suggests to us that more powerful and flexible learning-based approaches are needed.

Sage [44] uses a probabilistic distribution over metrics to identify the resources in a microservice that cause QoS violations. While its model can do the sort of flexible reasoning we target, it employs a large neural network superimposed on a directed acyclic graph representing causal dependencies between microservices. It is unclear how to adapt this model to handle cycles. Cycles are the common case in our target environments, even microservice-based environments (§ 2.2.2). Using data like the enterprise monitoring telemetry we work with – where there are numerous cycles amid hundreds of applications of essentially arbitrary functionality – there is no clear way to produce an acylic graph. As we show later, not handling cycles results in not being able to model all relationships which ultimately makes Sage incapable of producing the right root cause (§ 2.6.1), so it is effectively inapplicable in such environments.

Provenance-based troubleshooting systems [31, 95, 53, 94, 86] track all events and the causality between them to find the root cause of a performance error using the event DAG. However, obtaining such detailed monitoring data requires request- or packet-level tracing which is not feasible in most enterprise networks.

2.2.4 Summary of goals and existing methods

In order to meet our goal of performance diagnosis for common enterprise cloud environments, we need a scheme that (1) utilizes common monitoring information, which implies

[7]ExplainIt also was designed to answer multi-node conditional correlation queries interactively posed by the user, which is an assisted rather than fully-automated diagnosis that falls outside our use case.

handling complex dependency topologies that lack detailed dependencies and contain many cyclic relationships; and (2) achieves high accuracy. To the best of our knowledge, past approaches do not achieve these goals. Yet we can draw two important ideas from past work. First, NetMedic's idea of using an arbitrary directed graph to model dependencies makes no assumptions about cycles, allowing us to model directed dependencies where we have them, and use bidirectional dependencies otherwise. Second, Sage's approach involves an idea of using counterfactual reasoning that offers a more principled way to avoid the assumptions of heuristic-based approaches, by phrasing diagnosis as a *what-if* question: *If I changed metric M_r to a certain value, what effect would that have on the problematic metric M_o?* However, for the reasons noted above, we will need entirely new algorithms to make use of these high-level ideas in our environment.

2.3 USING MURPHY FOR PERFORMANCE DIAGNOSIS

We describe a typical troubleshooting experience with Murphy (see Figure 2.2 for example inputs/outputs of the tool).

Let's suppose an incident is reported in the IT infrastructure about a client facing service *foo* experiencing performance degradation. IT admins can run Murphy providing, as input, a *problematic symptom* – a problematic metric M_o of an entity E_o they want to know the reason for. This could be, for example, high memory usage of a SQL server used by *foo*. The problematic symptoms can also be obtiained via other methods such as by finding all entity metrics related to an affected application that are above thresholds preset by operators or via automated tools such as [41]. As the identification of problematic symptoms is not central to our design, we refer to § 2.5.2 for a more detailed discussion.

For each problematic symptom (M_o, E_o), Murphy outputs a ranked list of root cause entities for that symptom. Additionally, for each root cause entity, Murphy produces a causal explanation chain tying it back to the symptom.

2.4 DESIGN

There are three parts to the Murphy system (Figure 2.2):

1. The first part constructs the *relationship graph* using data obtained from the monitoring system (§ 2.4.1).
2. For each problematic symptom in the input, Murphy's inference algorithm generates

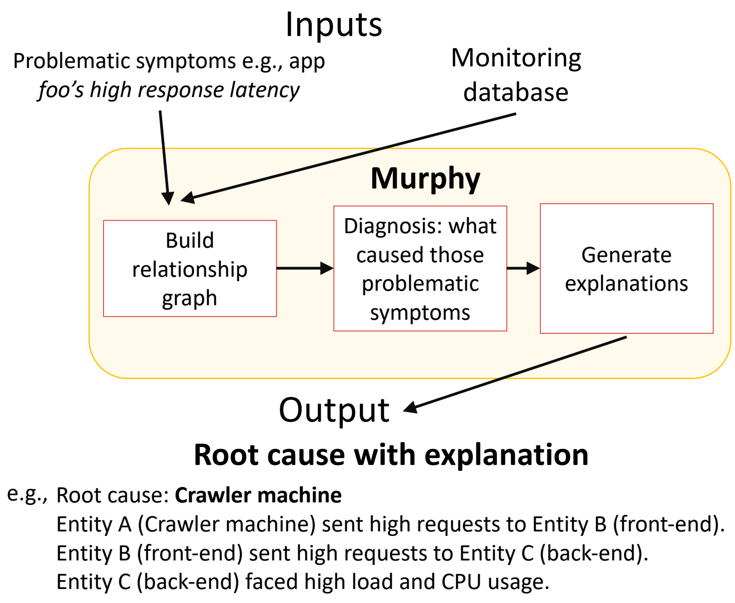

Inputs
Problematic symptoms e.g., app *foo's high response latency*
Monitoring database
Murphy
Build relationship graph
Diagnosis: what caused those problematic symptoms
Generate explanations
Output
Root cause with explanation
e.g., Root cause: **Crawler machine**
Entity A (Crawler machine) sent high requests to Entity B (front-end).
Entity B (front-end) sent high requests to Entity C (back-end).
Entity C (back-end) faced high load and CPU usage.

Figure 2.2: Murphy workflow

candidate root causes (§ 2.4.2) using a Markov random field (MRF). The MRF models a joint probability distribution of all entity metrics and is learned via historical values.

3. Murphy generates explanations for the root causes tying them back to the problematic symptoms (§ 2.4.3).

Our core contribution is in the design of the MRF framework which lets us utilize commonly available telemetry while producing accurate results. We describe each part next.

2.4.1 Constructing the relationship graph

Murphy employs a *relationship graph*, where the edges between entities are based on simple "neighborhood" relationships that are pre-defined by the monitoring software and can be easily extracted from the input. For example, a flow has edges to its source/destination VM, a VM has edges to its host and NIC, a microservice has edges to the container it resides on, and so on. This neighborhood definition is deliberately loose in order to work with common monitoring telemetry that doesn't have information about causal DAGs. Note this means that the relationship graph may have cycles.

To construct the relationship graph, Murphy makes an initial query to the monitoring database to obtain descriptions of a set of entities S relevant to the problem. If the input to Murphy is an affected application A, then S is the set of all entities that the system considers to be *members of* A. If the input specifies a problematic entity e, then S is the singleton set: $\{e\}$. Starting from S, Murphy constructs the relationship graph recursively by exploring the neighborhood of S and updating S as $S = \text{neighbors}(S)$. If the relationship graph becomes intractably large, then optionally this exploration is stopped after a few iterations. Figure 2.1 shows the relationship graph for a real incident. For each entity in the relationship graph, we also have timeseries data for various metrics (e.g. CPU/memory/network usage for a VM, session count and bytes sent/received for a flow, etc.).

By default, to be conservative, we add directed edges in both directions between two neighbor entities A and B in the relationship graph. The directed edges represent potential dependencies in both directions A$\rightarrow$B and B$\rightarrow$A. If a directional dependency is known between two entities, such as in the case of caller and callee microservices [44], then Murphy can incorporate that via a single directed edge. This design ensures that entity dependencies can be captured in the most general way and allows Murphy to work with commonly available telemetry information that may not include information about causal dependencies.

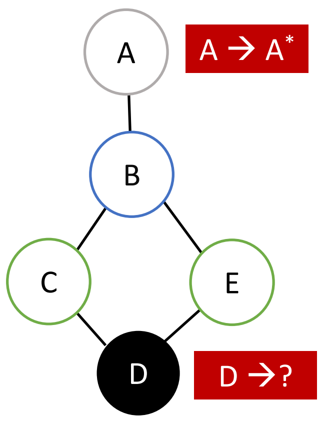
2.4.2 Performance diagnosis

For each problematic symptom (M_o, E_o) provided as an input (§ 2.3), Murphy separately runs the performance diagnosis algorithm.

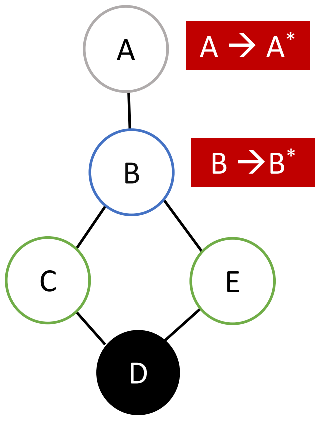
The relationship graph lends itself naturally for performance diagnosis via an appropriately designed graphical model. Markov random fields (MRFs) are one such family of probabilistic models. A MRF is a probability model superimposed on a graph that may have cycles, making it suitable to reason about entities in a relationship graph (§ 2.4.1). While acyclic causal edges make Bayesian networks easy to understand, a MRF is harder to interpret making it more challenging to apply. For the same reason, inference algorithms for Bayesian networks [25] don't apply to MRFs.

We've designed a MRF framework, according to the needs of our environment (§ 2.2.1), with available telemetry and its scale as the focus. We describe the MRF framework in two sub-parts: (a) model and (b) inference algorithm.

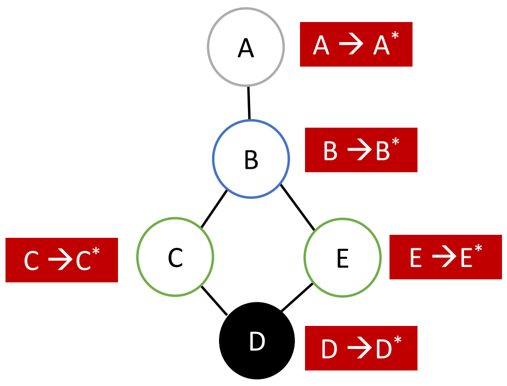
Model: To reason about a problematic (entity, metric) symptom, provided as input (§ 2.3), Murphy models the distribution of metrics for all entities as a MRF; call this distribution P_G. P_G denotes the joint probability of all entity metrics taking certain values and is calculated using a general directed relationship graph, that can potentially have directed cycles. P_G is



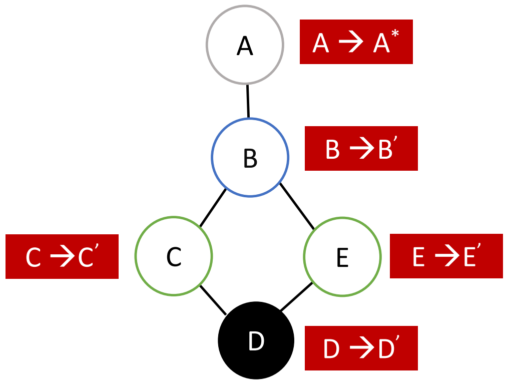
(a) Step 1



(b) Step 2



(c) Step 3



(d) Step 4

Figure 2.3: Inference algorithm- evaluating whether A is the root cause for D's high metric value for a toy relationship graph with only one metric per entity. For clarity, we draw an undirected edge to represent a directed edge in both directions. We (a) change A to a counter-factual value A^*, (b) resample B assuming the new value A^*, (c) resample C, D and E in a similar way, and (d) resample C, D and E, again g times for Gibbs sampling. We run the same resampling algorithm, this time starting with the current true value of A, instead of a counterfactual, and obtained the resampled value D''. After this sampling process, if $D' \ll D''$, then we classify A as a root cause for the high value of D.

defined as a product of individual entity factors P_v for every entity v:

$$P_G = \frac{1}{Z} \prod_{v:V(G)} P_v(v|\text{in\_nbrs}(v)) \tag{2.1}$$

In the above,

- $V(G)$ denotes the entities in the relationship graph G.
- in_nbrs(v) denotes the set of neighbor entities w such that there is an edge from w to v in the relationship graph.
- P_v is a function that takes as input, the values of metrics of v and v's incoming neighbors (in_nbrs(v)), and outputs a probability score between 0 and 1.
- Z is a (unknown) normalizing constant that ensures the probabilities add up to 1. The inference algorithm does not require the value of Z.

The relationship between an entity's metrics and its neighboring entities can be complex and variable across entities. Hence, the function P_v is determined by relating metrics of entity v in a time slice to the metrics of the neighbors of v in the same time slice. Specifically, Murphy learns a multivariate distribution for P_v, for all v, using a standard model such as linear regression with normal error, Gaussian mixture model (GMM), neural networks or SVMs using historical metric values. Different models could be suitable in different environments. The right choice of model can be determined by analyzing training errors in learning P_v across several entities v. In our production environment, we found ridge regression (a form of robust linear regression) to work best, evaluated using a large real world dataset (§ 2.6.6.1). Hence, we employ Ridge regression in Murphy for all our experiments.

Model training: Murphy doesn't keep any pre-trained models; every time Murphy is called, online training is triggered. Training online on fresh data has three advantages over training offline: (a) Applications get updated frequently, e.g. the application topology or software version might change from time to time. Using a model trained on outdated application topology or outdated software may not be ideal for diagnosis. (b) Online training eliminates the inconvenience of storing and maintaining a large number of entity models P_v. (c) Most importantly, an operator will run Murphy in the middle of an incident, so, with online training, the last few data points in the training data are from during the time of the incident. This turns out to be crucial (§ 2.6.5.1) as often an incident involves a pattern of metrics which hasn't occurred in the past (§ 2.6.2).

Every time Murphy is called, we train the linear regression model for each P_v using data from one week prior to the incident, which in our environment constitutes of a few hundreds time points for training. We didn't exhaustively explore all possibilities for the training period length, but found one week to be reasonable given using older data for training might include stale patterns from older app deployments as apps constantly get updated (also see § 2.6.5.2). A valuable area of future work would to be fully explore the tradeoff between using more data vs. fresher data, which could even depend on the specific incident.

Using a large number of features may cause overfitting when the training data is small. Hence, guided by the "one in ten" thumb rule [8] for regression problems (use at least 10 observations/parameter), we pick the top $B = 10$ neighbor metrics, based on their correlation with v's metrics (number of parameters is also B in a linear regression model). We also tried $B = 5$ and $B = 20$ and found training error to be within 3% of $B = 10$.

Inference algorithm: The distribution P_G provides a powerful way to reason about entities and their states. We describe the algorithm using a toy example, shown in Figure 2.3. Let's say we want to determine what caused a high CPU utilization at server D. To evaluate if another entity A in the relationship graph (not necessarily D's neighbor) might be responsible, we change the value of A's metric to a "counterfactual" value A', keeping the value of all other entities unchanged. Using a Gibbs-sampling like algorithm, we then resample D via the MRF to get a new value D', starting from the "counterfactual" value A' and $\mathcal{E}$, where $\mathcal{E}$ represents the original value of all entities besides A and D. We note that for resampling D, the sampling algorithm will have to first resample other entities via which A would affect D. We refer to this sampling algorithm as $P_G(D|A', \mathcal{E})$ which is described in the next paragraph. Intuitively, if this new CPU utilization D' is less than D, we can conclude that A is a contributor to the high CPU utilization at D. This probabilistic "what-if" analysis is referred to as the *counterfactuals* [49] technique. We use this idea to find out which entities can alleviate a problematic symptom (a problematic metric of an entity). Note that Murphy runs this inference algorithm on the current metric values whereas the training happens on the prior one week's metrics.

The last piece in the algorithm is the sampling method to sample D from $P_G(D|A, \mathcal{E})$ for entities A and D, which are not necessarily neighbors. Gibbs sampling is commonly used to sample from a MRF [68], but it entails executing several iterations of "pick a random entity and resample it given its neighbor entities". There are two problems with running exact Gibbs sampling. (a) It would be computationally too expensive in our environment since the relationship graph could have several thousands of entities. (b) We want to preserve the values of entities that are likely unrelated to A but might affect D. Running the sampling algorithm on those entities might destroy their values. Instead, we resample only a subset of entities that are on paths from A to D. We thus use a variant of the Gibbs sampling algorithm (illustrated in Figure 2.3 and detailed below).

Putting all the pieces together, the algorithm is as follows:

1. We set the value of A's metric to a lower/higher counterfactual value: A', that is 2 standard deviations away from its current value.

2. We consider the entities in the shortest path subgraph from A to D, defined as $\mathcal{T}$. For each entity v in $\mathcal{T}$, ordered in increasing distance from A, we resample the metrics of v from $P_v(v|\text{in\_nbrs}(v))$. The last step of this process, therefore, gives us a new sample value for D.

3. We repeat step (2) for $W = 4$ iterations (see § 2.6.6.2 on how we picked W). Sampling nodes more than once, as in Gibbs sampling, helps to propagate effects across cycles in the graph (§ 2.6.6.2).

4. Let d_1 be the sampled value for D thus obtained after step (3), having started with the counterfactual value A' in step (1). We also run the procedure (2)-(3), but this time start with the current true value of A, instead of the counterfactual value A', obtaining sampled value d_2. Then, we generate many such samples (5,000 in our implementation) for each of d_1 and d_2 via steps (1)-(3). If the d_1's are significantly less than d_2's (decided via a T-test), we conclude that A is a root cause for D. In our implementation, we generate 5000 samples for d_1 and d_2 each for the T-test.

We note that Gibbs sampling helps in propagating newer values across cyclic dependencies in the model. Let's say we need to resample a set of entities $\mathcal{T}$. Consider a cyclic dependency $A \to B \to A$ where A, B are entities in the set $\mathcal{T}$. If we only resample entities in S once and say we sample A before B, the newer value of B does not propagate across the dependency from B to A. Resampling A and B multiple times, as done in Gibbs sampling, solves this problem partially and improves accuracy (§ 2.6.6.2).

For each entity in the relationship graph, Murphy evaluates if its a potential root cause using the above algorithm. Murphy limits this search space of potential root cause entities via the following method: it runs a breadth first search starting from the problematic entity, exploring neighboring entities that have metrics above very conservative thresholds, while pruning out the rest. This reduces running time and improves precision. For fairness, we provide this pruned search space to all reference schemes that we compare with in our evaluation (this improved their accuracy).

Ranking the root causes: Once Murphy's inference algorithm produces the root cause entities, we rank them based on how anomalous their current metrics are. To do so, we

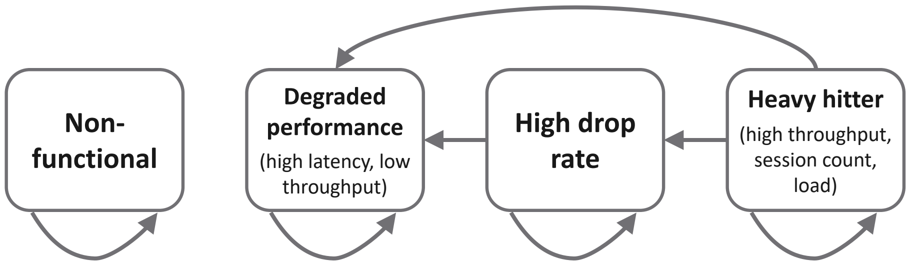

Non-functional
Degraded performance
(high latency, low throughput)
High drop rate
Heavy hitter
(high throughput, session count, load)

Figure 2.4: State machine encoding causal rules between entities' states for generating labeled explanation chains

consider how many standard deviations away a metric is from its historical mean value, which translates to a score for a single metric of an entity. We set the entity's score to be the score of its most anomalous metric. The ranking between the root cause entities is inversely proportional to this score.

Correlation vs causation: Note that as in [61, 63], the resulting candidate root causes have been determined to be *correlated* with the problem, but *causality* has not been determined. In the absence of precise information about causal dependencies between entities, Murphy does not guarantee causality between the root cause and the observed problems, but the candidate "shortlist" of potential root causes it outputs is still useful (as we will show later via experiments), since correlation is a necessary condition for the failure types within scope of Murphy.

Edge cases: Historical metric values may be missing for a newly introduced entity. To construct the training set for our algorithm in such a case, we use a default metric value (such as 0% for CPU usage) as a placeholder for missing values. Murphy also presents all recent configuration changes to the operator to catch problems caused by recently spawned VMs. Although the MRF framework captures a wide range of practical cases such as high CPU usage of a VM, high drop rate of a NIC, high latency of a service etc., other failure types are outside Murphy's scope (see § 2.7).

2.4.3 Explaining the diagnosis

Once Murphy finishes diagnosis and the candidate root causes have been found, we also generate human-readable explanations for them. We first assign one of the following labels to each entity based on their current metrics and conservative thresholds[8]: Faulty/Non-

[8]25% CPU/memory/disk/port utilization, 0.1% drop rate, 50 TCP sessions or 1GB byte count for a flow in a single time interval.

functional; Degraded performance; High drop rate; Heavy hitter; Okay.

We encode prior domain knowledge about causal truths using a state machine (Figure 2.4). Each node is a possible label state and the arrows indicate causal truths e.g. "Heavy hitter flow can cause high drop rate on a virtual NIC" or "Heavy hitter flow can cause high load on a VM". Although simple, such a labeling scheme produces semantically meaningful explanations for each root cause. Once we've decided the entity labels for every entity, we trace paths from root cause to affected application entities in such a way that each edge in the traced path respects the label causality rules described above. Note that this step does not affect the selection of root causes, and hence does not affect accuracy. We found it was convenient to provide plausible intuition for those root causes.

2.5 IMPLEMENTATION AND SETUP

We implemented Murphy in Python with ∼7K LOC. For reference schemes, we used the author-provided implementation for Sage and our own implementation of NetMedic and ExplainIT as their code wasn't available publicly. We test the schemes in two environments: (a) a cloud environment of a large enterprise running many production applications and (b) microservice-based applications (from the DeathStarBench suite [45]) running on private servers and a public cloud environment (AWS).

2.5.1 Setup and datasets

For our evaluation, we utilize datasets from two environments.

2.5.1.1 Datasets from apps in production environment:

We utilize two real world datasets that we collected from a commercial network observability platform (§ 2.2.1) monitoring the production infrastructure of a large enterprise.

2.5.2 Identifying problematic symptoms

A trouble ticket may not directly specify a problematic symptom in the form of an entity metric pair (M_o, E_o). How do we get from a ticket to a problematic symptom? In many cases, operators are able to identify troublesome symptoms in an application e.g. high user response times for a client-facing microservice, high resource utilization of a backend machine, high drop rate at a VM etc.. Operators can then specify problematic (entity, metric) symptoms to Murphy that they want to find reasons for.

Optionally, this step can be skipped and Murphy will find problematic symptoms on its own by scanning the affected application's entities, looking for anomalous metrics in the current time slice using preset thresholds [9]. Alternatively, one could also use other automated tools such as Revelio [41] to identify problematic symptoms that Murphy can diagnose, we leave this for future work.

For each problematic symptom, Murphy separately runs the inference algorithm for performance diagnosis.

Incident dataset: We describe our experience with a dataset of 13 real incidents (Table 2.2), with varying complexity and resolution times ranging from a few minutes to a few hours.

For each incident, we collected data for entities up to four hops away in the relationship graph from the affected entities (e.g. "all VMs of application foo").[10] We extracted the entities involved in the resolution from the trouble ticket, and treat that as the ground truth. We note however that this human-operator-decided ground truth may not always be the true root cause (e.g. the root cause was heavy load caused by a flow session that originated elsewhere but went away after rebooting all application VMs, and the reboots were stated as the resolution).

Metrics dataset: We collected metrics of $\sim$17K entities associated with over 300 production applications, for a period of a week. This data on its own is not sufficient to evaluate diagnosis accuracy as it does not come with information about failures. But, we can use it to run micro-benchmarks to evaluate model training accuracy, test various subroutines of the algorithm and fine-tune Murphy's algorithms on large scale production data.

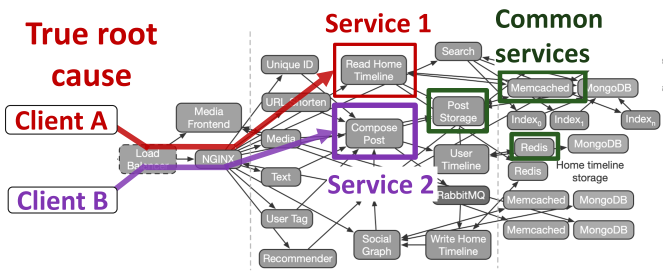
2.5.2.1 Datasets from microservices in public clouds:

We ran two microservice apps from DeathStarBench [45]:

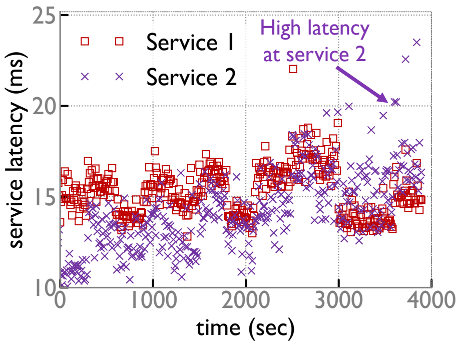
- Hotel-reservation on a dedicated 7-node Kubernetes cluster hosted on AWS (across multiple availability zones in *us-east-2*) with each node provisioned with 4-core Intel(R) Xeon(R) series CPU, 16 GB of RAM, 32 GB of SSD, and up to 5 Gbps bandwidth capacity.

[9]conservative thresholds: 25% CPU/memory/disk/port utilization, 0.1% drop rate, 50 TCP sessions or 1GB byte count for a flow in a single time interval. In enterprise environments, operators often configure such thresholds to receive alerts. Those thresholds could be used too.

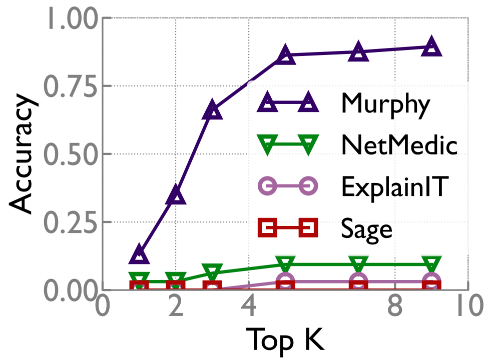
[10]As this environment is a private cloud, both virtual and physical entities are visible to the enterprise's monitoring platform and are included in the relationship graph in this data set.



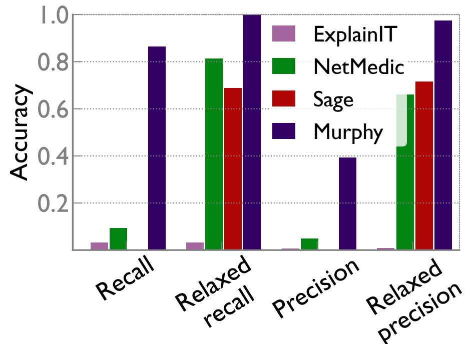
(a) Cause: Performance interference



(b) Symptom: Latency spike



(c) Top-K Accuracy



(d) Precision and recall

Figure 2.5: Performance interference experiment in DeathStarBench (§ 2.6.1). (a) Fault scenario: client A sends a lot of requests to service 1 which overwhelms the downstream common services shared between service 1 and 2, causing high latency for service 2 for Client B. "Common containers" are containers that "common services" reside on. (b) The fault gets injected just after 3000 seconds when Client A begins to send a lot of requests. (c) Accuracy (recall) in top-K. (d) Precision and recall (see § 2.6.1 for definition) of various schemes in producing the true root cause. Also shown is a relaxed notion of accuracy which measures how often a scheme gets at least one entity in top 5 that's either the true root cause or a common container or a common service.

- Social-network hosted on a single-node environment on a private cloud. The node was provisioned with 8 Intel(R) Xeon(R) series CPUs and 32 GB of RAM. The microservice applications are orchestrated using Docker with all inter-container traffic traversing through localhost.

The hotel-reservation app comprises 8 services and 16 total relationship graph entities including containers and services. The social-network app comprises 24 services and 57 total entities. We obtain metrics of the application entities from two sources, viz., (1) container metrics like average CPU/memory/disk/network usage from Cadvisor [1], aggregated over 10 second intervals and (2) microservice service latencies by aggregating the individual response latencies, also over 10 second intervals. We use wrk2 [11], an open-loop workload generator to send requests to the application. We get the request traces via Jaeger [6].

We create multiple types of failure scenarios in these microservice environments:

- **Performance interference**: We set up two clients, A and B, who send requests to two different API endpoints, service 1 and 2 respectively. The API call trees of service 1 and 2 share some common backend services as shown in Figure 2.5a. Client A generates a high request load, overwhelming a subset of these shared downstream microservices. As a result, response latency of service 2 increases (see Figure 2.5b) impacting the latency observed by client B. The problematic symptom that we provide to the tool to diagnose is client B's observed latency and the true root cause is high RPS load of client A. We generate 32 variants of such scenarios by changing the RPS load sent to the services. This failure scenario was motivated by the production incident mentioned in § 2.4 (Figure 2.1).

- **Resource contention**: using `stress-ng`, we inject CPU, memory and disk faults to randomly chosen application containers, as in [44]. We generate more than 200 such fault scenarios across both the setups, varying intensity, duration (5-10 mins) and location of the faults while client workload (30-90 minutes long) is in progress.

2.6 EVALUATION

Evaluation goals: The goal of our evaluation is to investigate Murphy's diagnosis accuracy compared to reference schemes Sage, NetMedic, and ExplainIT. We feed the same input relationship graph to all schemes when possible (i.e., if the algorithm can take it as input). We first evaluate scenarios in our enterprise environment and an emulated microservices environment which commonly have cyclic dependencies. Second, to enable comparison with Sage, we also consider a restricted set of scenarios where there are no cyclic dependencies

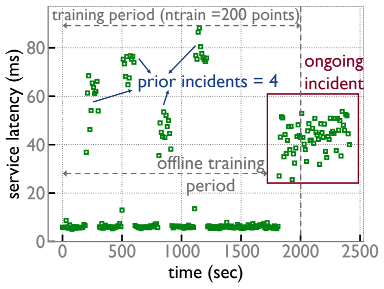
Incident (observed problems)	Murphy FPs	NetMedic FPs	ExplainIT FPs
1. Two apps nodes crashed due to a plugin	6	69	93
2. App returning a 502 error	0	1	0
3. App unavailable	4	40	60
4. App slow, experiencing timeouts	10	4	4
5. App unavailable	1	1	1
6. App redirecting to a maintenance page	4	1	1
7. Heap memory issue with a node	1	1	1
8. App performance degradation	6	67	189
9. App failing with 503 error	1	1	1
10. Health check failing on 2 nodes	2	2	23
11. App redirecting to a maintenance page	6	10	22
12. Slowness in loading data	20	101	21
13. Performance alert about a node exceeding thresholds	0	0	0
Average false positives	**4.9**	**23.2**	**32.3**

Table 2.2: Number of false positives (FP) produced by each scheme for each incident, according to operator decided resolution (see § 2.6.2), for incidents in the dataset. Only FPs are shown, rather than false negatives, because the schemes were calibrated to have similar false negatives (§ 2.6.2).

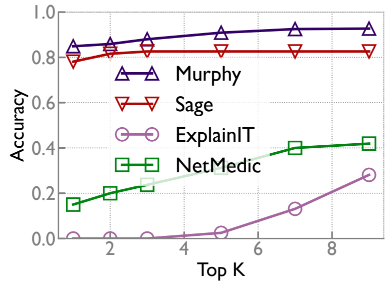
between entities as done in [44]. We evaluate how robust each scheme is in handling degraded data with omissions/errors, which can be present in common telemetry. We then show several microbenchmarks to quantify the effect of the design choices and algorithmic subroutines of Murphy. Finally, we discuss the runtime performance and the sensitivity analysis.

Measuring accuracy: We measure *Top-K* accuracy, equivalently called *recall*, defined as the fraction of times the true root cause entity is among the first k entities in the ordered list of candidate root causes generated by the scheme. We use K=5 unless stated otherwise.

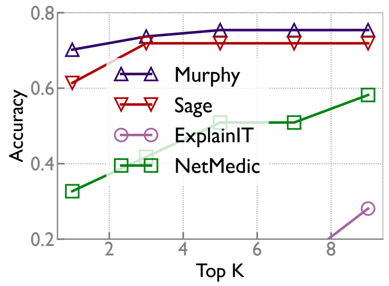
We also show *precision*, defined as either $1/r$ if the scheme outputs the true root cause as the rth candidate, or 0 if the output does not include the true root cause at all. The intuition for this is the operator will start at the top of the list and will have to check r suggestions before finding the right answer, and any false positives ranked beyond r won't matter.



(a) A sample latency trace



(b) Top-K Acc (social-network)



(c) Top-K Acc (hotel-reserve)

Figure 2.6: Resource contention experiment on DeathStarBench applications: (a) service latency in a sample trace across time. The training period contains 4 prior incidents while the main incident is triggered at t = 1800s. (b), (c) Top-K Accuracy (Acc) for varying K.

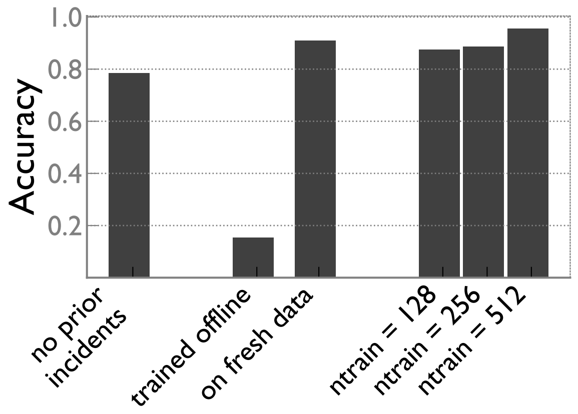


Figure 2.7: Various microbenchmarks with Murphy. Fresh data implies that Murphy was trained with data that included several minutes during which the incident was in progress.

2.6.1 Performance interference in microservices

We consider the performance interference failure scenarios described in § 2.5.2.1 where high load at service 1 overwhelms the common downstream microservices it shares with service 2, causing high latency for service 2. To model the effect of the two services on each other, the relationship graph should have a path from service 1 to service 2 and vice-versa. This however induces a cycle in the relationship graph. Sage's model can't handle such cycles, and hence only models a single user-facing service and its downstream services. As a result, the true root cause (service 1) falls outside its model, preventing Sage from catching it.

Figure 2.5c shows Top-K accuracy for varying K and Figure 2.5d shows the precision and recall for K=5. Murphy produced the true root cause in the top-5 86% of the times, while Sage, on account of the true root cause being outside its model, did not produce the true root cause (service 1 in our example) in any case (i.e. 0 recall). Other schemes also did not produce the true root cause most of the times (accuracy $< 15\%$).

Can Sage get *close* to the root cause, while working within the scope of its model? Specifically, identifying the common containers that are overwhelmed may be a helpful step towards the true root cause. To test this, Figure 2.5d also shows a very relaxed notion of recall: a scheme achieves 100% recall if its top-5 contains at least one entity that is either the true root cause, a common container or a common service. Relaxed precision is defined similarly: it is inversely proportional to the number of false positives seen by the operator before one of the "relaxed"-root causes is produced by a scheme. Murphy achieves perfect relaxed-recall while NetMedic also has good relaxed-recall of 0.81. Murphy has significantly better relaxed-precision and relaxed-recall than Sage, NetMedic and ExplainIT.

2.6.2 Incidents in production environment

Table 2.2 shows the number of false positives (FPs) produced by NetMedic, ExplainIT and Murphy for each of the 13 incidents in the incident dataset (§ 2.5.1.1) from the production environment. The table shows only FPs, because the schemes were calibrated to have similar false negatives.[11] Sage is incapable of working in this environment as it requires a causal

[11]We calibrated each scheme's parameters to minimize false positives under the constraint that they produce recall = 1 (equivalently, zero false negatives) on a certain set of "calibration incidents". On the full set of 13 incidents, recall was not quite identical across all schemes, but was very close– all schemes had a recall in the range $[0.53, 0.56]$.

The calibration incidents were the 2 incidents for which we had full certainty in the ground truth via discussions with operators. Recall that in general, we took ground truth to be the entities involved in the operator's resolution of the incident, which in some cases may not be the true root cause.

DAG of dependencies which we don't have. This major limitation prevents us from using Sage in our production environment.

As can be seen from Table 2.2, Murphy overall produced 4.7x fewer false positives than NetMedic and 6.6x fewer than ExplainIT. We observed that for some incidents, both NetMedic and ExplainIT produced many false positive root cause entities that were highly correlated with the problem while Murphy was able to prune them out (incidents 1, 3, 8 and 12).

We remark on some incidents below, including discussions with network operators about the utility of Murphy's analysis:

- For incident 2 in Table 2.2 (illustrated in Figure 2.1), Murphy correctly identified the root cause entity. We validated both the root cause and the explanation produced by Murphy by talking to network operators. The top explanation chain produced by Murphy for this incident was:
  - Heavy-hitter flow from crawler VM (true root cause)
    $\rightarrow$ Front-end VM
    $\rightarrow$ Heavy hitter flow
    $\rightarrow$ High CPU on backend VM
- In one incident, 2 nodes failed health checks. Murphy flagged flows that were sending high traffic to the nodes. However, the operators rebooted the nodes to resolve the incident, so the operator-decided ground truth did not include the flows.
- In another incident, operators were unable to pinpoint the root cause of an incident that lasted only for six minutes. Interestingly, a similar incident occurred after a few weeks. Murphy flagged two flows, which were likely due to network upgrades, as culprits.

2.6.3 Resource contention in microservices

We consider the resource contention failure scenarios (§ 2.5.2.1) which don't have any cycles. Sage was designed for such scenarios [44]. Figure 2.6a shows the response latency in a sample scenario. For realism, as in [44], we induce up to 14 "prior incidents" where short-lived faults are injected on randomly chosen containers before the actual incident. Refer to § 2.6.5.3 for accuracy when there are no prior incidents.

Figures 2.6b, 2.6c show the accuracy in producing the right root cause in the top K for varying K on the x-axis. Murphy produced the true root cause with high accuracy (overall, 77% as the top candidate and 83% of the times in top-5) and had a somewhat higher accuracy than Sage (69% in top-1 and 77% in top-5). ExplainIT looks at the entity whose metrics are most correlated with the high latency of the client and ends up producing entities that are closest (in the microservice communication graph) to the problematic entity such as the front-end container. NetMedic did not perform adequately, likely because of assumptions in its ranking heuristics: we found that its geometric mean based path weights can produce entirely unrelated subtrees of entities in the microservice graph.

2.6.4 Accuracy with incomplete data

Table 2.3 shows accuracy when the data is "corrupted" with some omissions or errors. Such errors can exist in the monitoring data for large infrastructure and hence a diagnosis scheme should be robust to them. To ensure a comparison with Sage, we use the same setup as § 2.6.3 with no cycles. We evaluate four such cases where we introduce errors in the monitoring data:

- Missing edge: we remove the association between a randomly chosen RPC and its parent (caller) RPC
- Missing entity: we remove a randomly chosen entity, including all its metrics and associations
- Missing metric: we remove a single metric (e.g. memory usage) for the root cause entity
- Missing values: for randomly selected 25% of the entities, we remove their historical values (except for the values during the incident itself, which is still present)

Such errors could arise from bugs in the tracing framework (missing edge), or missing coverage in monitoring (missing entity/metric) or a newly spawned entity (missing values). As can be seen from Table 2.3, both Murphy and Sage are fairly robust, incurring 6% and 10% loss respectively. Missing values have a minimal effect on Murphy since the most recent data related to the incident is still present (see § 2.6.5.3 and § 2.6.5.1); it affects Sage significantly likely because its neural networks require more data points to learn the right pattern.

2.6.5 Microbenchmark experiments

We describe "microbenchmark" experiments where we evaluate various aspects of Murphy's design.

Scheme	Missing values	Missing edge	Missing entity	Missing metric	Aggregate (avg(1-4))	Unchanged input
Murphy	0.84	0.75	0.78	0.81	0.80	0.86
Sage	0.64	0.67	0.66	0.82	0.70	0.80
NetMedic	0.16	0.20	0.16	0.22	0.18	0.22
ExplainIT	0.05	0	0	0	0.01	0

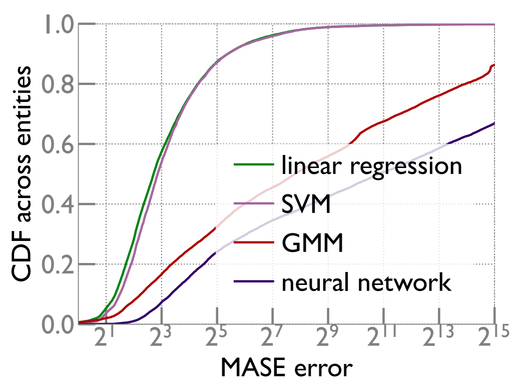
Table 2.3: Robustness: Accuracy with degraded/incomplete data. Numbers show recall in top-5. ExplainIT was designed for a different use case – as an interactive tool for queries posed by the user, hence its accuracy was low for automated diagnosis. Both Sage and Murphy were fairly robust.

2.6.5.1 Online vs offline training:

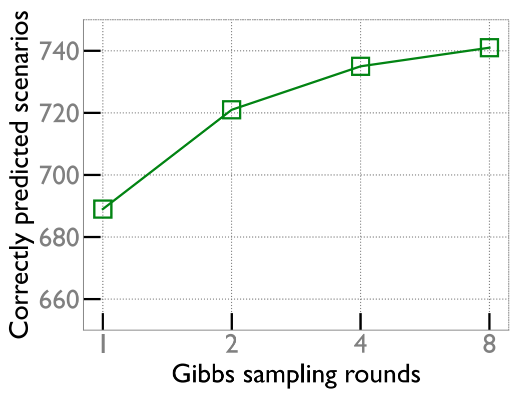
Murphy learns the distribution online from past metrics of one week so that the training data includes some recent data points when the incident has happened. Another alternative could be to train the system offline, so that training time is not a concern and potentially more training data can be used (as in [44]). However, as the bar labelled *trained offline* in Figure 2.7 shows, not including the incident data points results in a drastic drop in accuracy from 90% to 15% on the resource contention scenarios (§ 2.5.2.1). To aid offline training, we used scenarios with maximum prior incidents (=14). This poor accuracy on debugging incidents not seen before drives our design choice to train Murphy online, every time it's called, on the latest data. Although Sage originally was designed to be trained offline, for fairness, we also train Sage online which yielded higher accuracy for Sage.

2.6.5.2 Effect of length of training data:

The last 3 bars of Figure 2.7 show Murphy's accuracy with 3 different lengths of training durations on the setup in § 2.6.3. Murphy's accuracy improves significantly from 87% with 128 points to 95% with 4x more training data. There's a tradeoff between using longer training data and the running time since the training happens online in Murphy, when the tool is run by an operator. We found using the prior one week of historical data to be a good tradeoff point in our production environment (§ 2.6.2). The application characteristics like topology, configs, application version, workload, etc., also change from time to time, which is another reason to train only on recent metrics.



(a) Error in metric prediction for 17K entities



(b) Multiple iterations of Gibbs sampling improves accuracy

Figure 2.8: (a) Errors in predicting the metrics of an entity, given the metrics of all its neighbors across 17K entities spanning 300 production apps. (b) Gibbs sampling improved accuracy of prediction across multiple hops, consistent with the existence of cyclic effects in the production environment.

2.6.5.3 Accuracy with no prior incidents:

Often, the root cause for a problematic symptom involves a pattern of metrics that hasn't been seen before. Being able to reason about such scenarios is crucial for Murphy's usability. To test such cases, we generated 64 traces using a similar setup as § 2.6.3, each with no prior injected incidents. Since Murphy learns the distribution online, the training data still includes data from the current incident which needs to be diagnosed (see § 2.4.2). Figure 2.7, in the bar labelled *no prior incidents*, shows that Murphy correctly produces the true root cause 78% of the times in the top-5 and 62% of the times as the top root cause candidate.

2.6.6 Testing Murphy's internal components

2.6.6.1 Comparing metric prediction models:

Recall that Murphy internally reasons about root causes using a metric prediction model, that predicts how a change in one metric will affect others in the relationship graph. Here we test accuracy of the prediction model, using various methods of learning the model. Testing just this model doesn't require incidents, so we can leverage our much larger metrics dataset (§ 2.5.1.1).

Figure 2.8a shows the CDF of absolute error in predicted values across 17K entities from the metrics dataset, when using Ridge, Gaussian Mixture Model (GMM), SVM, and neural networks[12] as the metric prediction model. We found that Ridge linear regression, a variant of robust linear regression, works the best in our production environment. We believe neural networks pose a challenge because the number of available training data points is small (a few hundred).

2.6.6.2 Verifying existence of cyclic effects:

We wanted to evaluate if accounting for cyclic effects in the model improves the training accuracy in our production environment. Extensive testing in the real world is hard in the absence of the ability to run controlled experiments.

To do so, we design an experiment to measure the effect of flow entities on an entity that's multiple hops away from the flows- a backend SQL server. We picked from the metrics dataset of production apps(§ 2.5.1.1)- 24 applications that had at least one SQL server. For each application, we chose a randomly chosen backend SQL server Q, and use correlation scores with Q to pick top 5 flows, say flows' F, that sent requests to the front-end of the application. We then test if Murphy can predict the effect of changing metrics of flows in F, on the SQL server Q which is multiple hops away in the relationship graph (see figure 2.9). More precisely, we take two points in time t_1 and t_2 when Q had significantly different metrics. Keeping the metrics value of all entities other than F to be same as t_1, we update the metrics of the flows F to their values from t_2. We then run Murphy's resampling algorithm (§ 2.4.2) to obtain predicted metric values of the backend-SQL server- M^Q_{pred}. We measure if the predicted metrics M^Q_{pred} is "close"[13] to the real value M^Q at t_2. Figure 2.8b shows that running more than one round of Gibbs sampling in the resampling algorithm (§ 2.4.2) increases the number of scenarios correctly predicted by 5-10%. Since running more round of Gibbs sampling propagates cyclic effects in the relationship graph, this demonstrates that handling cycles correctly boosts the metric prediction accuracy and also confirms the existence of cyclic effects in our production environment.

2.6.7 Performance and handling scale

Murphy's inference algorithm has a runtime complexity of $O((N + M)T + (N + M)W)$ where N, M are the number of entities and edges respectively in the relationship graph, T

[12]We tried small neural networks up to 3 layers, with 5 neurons each.

[13]We define closeness using a criteria that allows for a small additive error and a constant multiplicative error characterized by constants ϵ and Δ, defined as: (Δ, ϵ)-criteria: if the predicted change is δ and the actual change is δ^*, we say that the algorithm is right if either $\delta^*/\Delta < \delta < \Delta.\delta^*$ or $|\delta - \delta^*| < \epsilon.V$, where V is the maximum value of the metric seen so far.

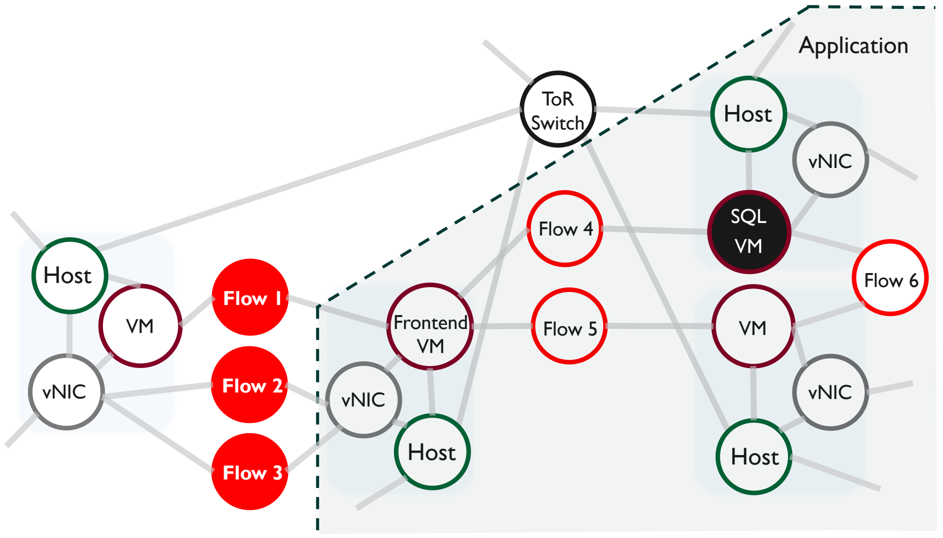


Figure 2.9: Experimental setup for verifying existence of cyclic effects. We change the values of the three flows in red and measure how well our Gibbs sampling algorithm can predict the corresponding change in SQL VM in black.

is the number of time slices in the monitoring data that the model gets trained on and W is the number of Gibbs sampling iterations. For our incidents dataset, N was typically a few thousands, M was roughly 10-20 times N, T was around 300 and W was 4. Two components contribute to Murphy's running time in our production environment. First, for the incident in Figure 2.1, it took less than a minute to fetch the metadata and metric values from the database for over 10 thousand entities – the typical size of the relationship graph. Second, Murphy took $\sim$2 minutes on average to produce the root cause entities for a problematic symptom, including the online training time. We can optimize this further with parallelism and by leveraging a C++/Java implementation (as opposed to Python).

2.6.8 Sensitivity Analysis

Murphy has some parameters, all of which can be tuned offline. We discuss Murphy's sensitivity to these parameters

- Gibbs sampling iterations (W): Figure 2.8b shows that increasing W led to improvement in accuracy and in general higher W would lead to higher accuracy by giving Gibbs sampling to converge more quickly. A higher W also means a higher running time, hence there's a tradeoff. As Figure 2.8b shows, the marginal benefit decreases with increasing W and so we settled on $W = 4$.

- Length of training data: We found a slight increase in accuracy with an increase in the length of training data. Refer to the last 3 bars of Figure 2.7 and § 2.6.5.2.

2.7 LIMITATIONS AND FUTURE WORK

While more advanced monitoring can be useful in some cases, e.g. to attribute the high request rate of a VM to a software bug, it's important to have effective diagnosis tools based on commonly available monitoring tools. Murphy's entity-level localization is already useful to operators; we leave diagnosis with advanced monitoring for future work.

Failures outside Murphy's scope: Although Murphy captures many practical cases such as high CPU/drop rate of a VM, high latency of a service etc., many scenarios are not covered, and could be the subject of future work. One such case is where cause and effect are separated in time, or metastable failures that persist even after the trigger is removed [58]. Murphy might not handle non-linearity in metrics (e.g. if load shedding kicks in after a threshold) since its implementation uses linear regression. Using a different learning model, such as neural networks, for Murphy's MRF framework might resolve this problem. Other examples include: failures that are local to an entity (process or thread), which may require finer-grained telemetry; software errors that don't manifest in any metrics and may be more suitable for program analysis [70]; and scenarios where the root cause is because of an aggregated metric of multiple metrics such as the combined session count of multiple flows.

Using Murphy for performance reasoning: Murphy's counterfactual reasoning framework was useful for performance diagnosis, and may also be applicable to other use cases. For example, it provides a way to evaluate the effect of a config change on a metric: e.g., how would the response latency change if allocated CPUs of the VM is increased by 2x? However, the required level of accuracy for this use case may be different than for performance diagnosis.

Leveraging offline training: While online training works well, a combination of offline + online training could still be leveraged; see § 2.6.5.1. This is common in the NLP domain [80] that has time-series data similar to ours.

2.8 SUMMARY

Performance diagnosis is a persistent challenge for enterprise infrastructure teams. In our work we found that making assumptions about the performance relationships between components or using fixed heuristics for inference harms diagnosis. Instead, we propose using MRFs and learning-based methods that do not make these assumptions for diagnosis in enterprise environments where relationships are rich and complex and evolve across time. We presented a possible design for such an algorithm in Murphy. Murphy not only meets the criteria for diagnosis in our production setup (by virtue of handling complex, cyclic dependencies) but also outperforms current diagnostic tools in their intended environments.

CHAPTER 3: FLOCK: ACCURATE NETWORK FAULT LOCALIZATION AT SCALE

3.1 INTRODUCTION

Datacenters often comprise of tens of thousands of network components. Failures in such large networks are common, arising due to software bugs, misconfiguration, and faulty hardware, among other reasons [59]. In many cases, a device will directly report a failure, e.g., a switch may report that one of its line cards is non-responsive or that an interface has a certain packet loss rate. These metrics are collected via monitoring software and used to raise alerts. However, datacenters also experience significant network downtime and SLO violations from *gray failures* whose root cause is obscure [59, 104]. For example, the reason for poor performance of a distributed service could be a link silently dropping a small fraction of packets without updating switch counters [90], or a driver bug in a virtualized firewall. Diagnosing such performance anomalies is very hard for network operators [24]. With programmable switches, more advanced monitoring is possible [103, 60], but these methods either do not eliminate gray failures [76] or come at a high cost in switch resources [103], and require deployment of programmable switches which is not generally available.

An alternate approach, which is the domain of this paper, is to infer the root cause via end-to-end measurements, which we refer to as *fault localization* [24, 90, 50, 13, 57, 47]. Fault localization has been deployed by large cloud providers [25, 50]. At the heart of fault localization is an *inference model and algorithm* that uses end-to-end observations (e.g., packet loss rate or latency of TCP connections) to infer a set of faulty components (links or switches). The key challenge is to do this both accurately and quickly.

The most powerful class of inference techniques builds a probabilistic graphical model (PGM) and performs a form of maximum likelihood estimation (MLE): finding the set of components that, if they failed, maximizes the probability of having produced the given end-to-end observations. An early such system was Sherlock [25], for inferring faults among dependent services. However, deriving the MLE for a PGM can be computationally intensive. With $\leq k$ failures among n components, the solution space is exponentially large in k ($O(n^k)$). In a datacenter with millions of TCP flows and links, and multiple simultaneous failures, Sherlock's MLE can require several hours.

Therefore, fault localization schemes for datacenter networks move away from PGMs to other techniques – using scores to rank links [24, 57] or obtaining drop rates via a system of equations [90]. As we will show, these compromise accuracy and flexibility of PGMs in favor of performance.

In this paper, we present Flock, a fault localization system for datacenter networks that seeks to maximize accuracy, with sufficiently high speed (i.e., seconds). Flock's core innovation is a novel MLE inference algorithm for PGMs that offers substantial acceleration for the kind of models encountered in fault localization, leveraging two key acceleration approaches: (i) a technique we call *joint likelihood exploration* maintains an array of hypotheses that it can update en masse to find the likelihood of a set of *new but similar* hypotheses, more quickly than computing their likelihoods individually from scratch and (ii) we use a greedy algorithm which builds its solution link by link; this part of the algorithm is simple, of course, but importantly, we prove a sufficient condition for optimality and verify through experiments that it does find the MLE in practice. These two optimizations each individually provide asymptotic speedups, and together allow Flock to use a PGM at scale. The result is that Flock is several orders of magnitude faster than past PGM-based fault localization [25], and is substantially more accurate than past non-PGM-based fault localization [24, 90], on the same input data.

Moreover, the PGM-based approach allows Flock to use different types of input telemetry. Recent datacenter fault localization schemes [90, 24] use observations of *active probes* of the network (which can be constructed to have known paths and uniform distribution) but do not incorporate *passive flow monitoring*, i.e., observations of all ongoing traffic, obtained via NetFlow, IPFIX, or INT [14]. The large volume of passive data makes it potentially informative. But including passive monitoring would be hard for non-PGM methods because it requires more discerning modeling and inference to handle skew in traffic patterns and path uncertainty.[15] Although PGMs are generally more flexible in incorporating data of different types, it would be hard for past PGM approaches because of computational difficulty, due to the immense number of flows and because a flow with 10 possible paths is roughly 10× costlier to model than a flow with a known path. Flock's combination of flexible PGM-based modeling and speed enables it to utilize passive information.

In summary, this paper's key contributions are as follows:

- **Inference algorithm.** We develop a new fast MLE inference algorithm for discrete-valued PGMs § 3.3.3 (the type of PGM used for fault localization). We analyze a sufficient condition for this algorithm's accuracy on Flock's model, providing intuition for why it works well in practice(§ 3.4.2).

- **System implementation.** We implement Flock (§ 3.3), a new end-to-end fault local-

[14][24] incorporates only a limited amount of passive monitoring; §3.2.3.

[15]Most data centers use non-deterministic ECMP multipath routing, so that only a *set of possible paths* is known when flows are monitored with NetFlow/IPFIX. Paths can be known with INT-based monitoring, but INT is not generally deployed.

ization system. The Flock algorithm forms the heart of the Flock system, allowing it to employ a PGM and naturally incorporate various kinds of dependence and uncertainty such as unknown paths.

- **Evaluation suite.** We create an open evaluation suite [15] for fault localization, which includes (a) implementations of algorithms from NetBouncer [90], 007 [24], Sherlock [25], and Flock, (b) an implementation of end-host telemetry agents and a collector, (c) telemetry data for six different fault scenarios from a simulated data center and a hardware testbed, and (d) scaling tests. We believe this suite is of independent interest, as it is the first such open data set and expands on the fault scenarios evaluated by past work.

- **Performance evaluation.** For a Clos network with 88K links and 9.5M flows, Flock is empirically $> 10^4\times$ faster than Sherlock's PGM-based method [25], scanning ~3.5M hypotheses in 17 sec, while achieving the same or better inference results (§ 3.7.8). In fact, Flock is $\approx 4.5\times$ faster than the non-PGM approach of NetBouncer [90] on the same input. 007 [24] is the fastest of the lot, but its time savings (<1 sec) is not a good tradeoff with accuracy.

- **Accuracy evaluation.** With the same (active probe) input measurements as past work, Flock reduced inference error by $1.8 - 8\times$ compared to 007 and by $1.19 - 11\times$ compared to NetBouncer.

- **The value of passive information/INT.** Incorporating passive monitoring reduced error even further, by up to $5.3\times$ compared to Flock with only active monitoring. We also evaluate the value of INT telemetry input.

3.2 BACKGROUND AND MOTIVATION

3.2.1 When fault localization is useful

Ideally, network faults are directly reported by the faulty component; e.g., switches typically track interface utilization, packet drops, up/down status, queue length, etc. These metrics are commonly collected from network switches via SNMP [29], polling [34, 87], or streaming telemetry [40, 23].

Fault localization becomes useful when such direct monitoring is not enough. A *gray failure* occurs when a faulty component does not report that it has failed [59]. For example, silent inter-switch or inter-card drops [103, 90, 24, 83, 104] are extremely challenging to detect, constituting 50% of faults that took >3 hours to diagnose in [103]. Other examples of gray failures include corruption in TCAM-based forwarding tables causing black holes [66, 104] or loss [35], and a misconfigured switch causing high latency [104] (see [103] for more cases). Gray failures also occur in the numerous software packet processing components present in modern data centers. Software bugs can silently drop or corrupt packets in host virtualization [91, 92], server software [104], and virtualized network functions like software firewalls [78]. All the above faults can be "silent" (the device does not realize the error occurred). Further, the symptoms could manifest at a different location than the problem, e.g. packet data could be corrupted by an intermediate switch but the corruption is discovered only at the receiver. End-to-end observations can help detect such problems (§ 3.6.4).

Another alternative is to use programmable switches to obtain more information, as in Omnimon [60], FANcY [73] (for ISPs), or NetSeer [103] which runs a packet sequencing protocol across neighboring hops to find silent drops. This can be quite accurate, though it comes at the cost of significant switch resources ($\sim$ 100% overall PHV usage and 40% ALU usage [103]). But more importantly, although use of programmable switches is growing, they still have very limited deployment (13% estimated market share for 2023 [52]). Both programmable and traditional switching environments are valuable use cases, but *the latter is the target of this paper;* schemes like [60, 103, 73] are out of the scope of our work. As an exception, we will consider the use of data collected with In-band Network Telemetry (INT) [76, 26], which does not directly report gray failures, but does record packets' paths. INT can be implemented with programmable switches, but similar path data can be obtained other ways; see § 3.6.2.

Thus, it is very hard to guarantee that every packet processing element detects all faults locally (indeed, the end-to-end principle [84] applies here). Even if a fault is reported, the operator may want a way to cross-check. For these reasons, we see fault localization based on end-to-end observations as an important tool in infrastructure engineers' toolbox for the foreseeable future.

3.2.2 Problem setup and goals

The input to a network fault localization algorithm is the network topology, and input flow telemetry. Each flow measurement includes one or more metrics (TCP loss rate, mean latency, throughput, etc.) and a set of paths through the topology that the flow may have traversed. Depending on the monitoring method, this set may have size one (the exact path is known) or greater than one (typically, a set of possible ECMP paths is known). Given this input, the fault localization algorithm should output a set of links or devices it believes to be faulty, while meeting two goals.

Performance: Network operators often have to resolve a reported problem quickly. For example, a managed service from BT has a 15 minute response time in its SLA [28] and Gartner chose a threshold of 3 minutes in defining "real time" network data analysis [22]. Thus, fault localization within minutes is critical and within a few seconds is ideal.

Accuracy: False positives can bury true problems among several alerts [82]. False negatives could send engineers down the wrong track of investigation. There may be a tradeoff between accuracy and performance. As long as results are available within a few minutes, accuracy (minimizing false positives and false negatives) is of primary importance.

3.2.3 Existing fault localization approaches

Several past approaches address above goals, with different types of *input telemetry* and different *inference algorithms*.

Input telemetry: Administrators commonly [22] use passive monitoring of flows via NetFlow [37] or IPFIX [38] to understand overall network health. However, this has not been relied on for automated fault localization because vendor-specific ECMP hashing obscures flows' exact paths.

Recent approaches use active probes with known paths. NetBouncer [90] sends probes uniformly from hosts to core switches in a Clos topology, via special switch support. 007 [24] uses active probes with assistance from passive monitoring: end-host agents monitor production traffic, and on detecting flows with anomalous performance, traceroute the flagged flows' paths and report the flagged flows' metrics for analysis. 007 does not incorporate passive monitoring of non-flagged flows. In both systems, the volume of active probes is limited (which assists algorithm runtime, and minimizes host/network overhead), and the exact path of monitored flows is known (which assists accuracy). But active probes do not eliminate all uncertainty: even if the path is known, the culprit link(s) are not; and flows can experience packet loss or latency on non-faulty links (e.g., due to congestion). Hence, good inference is still needed. Further, active probes often take a different data path than regular traffic and may fail to reproduce problems faced by production flows [24].

INT has, to our knowledge, not been specifically used by past work for end-to-end fault localization. INT is similar to passive NetFlow telemetry in that it can observe a large volume of actual application traffic, if deployed for all flows. However, like active probes, it can trace the traversed path. It also does not eliminate all uncertainty (e.g., a silent packet corruption will not trigger an INT action, and even if the path is known, the faulty link is not).

We observe that *different deployments are likely to have different available information.* 007 requires host support and NetBouncer and INT require switch support. INT is now available in some switches, but is not deployed in most networks, and might be deployed selectively. Furthermore, this technology landscape may evolve. A flexible scheme in terms of input telemetry is thus preferred.

Inference algorithms: Sherlock [25], NetSonar [97] and Shrink [62] use a PGM with a form of maximum likelihood estimation (MLE), but are far too slow. Sherlock targeted a small use case (358 components), and NetSonar, which uses Sherlock's inference, targeted smaller inter-DC networks. In a data center network with tens of thousands of components, they require several hours (§ 3.7.8), even with $K \leq 2$ concurrent failures (they scan $O(n^K)$ possibilities, where n = number of components). Furthermore, inevitable concurrent failures [90, 24, 83] may make $K > 3$ essential.

Hence, state-of-the-art data center fault localization schemes move away from PGM-based MLE. 007 [24] uses a scoring function to rank links while NetBouncer [90] optimizes an objective function to solve for drop rates. These are reasonably fast, but as we will see, they fall short on accuracy.

Summary: Our goal is to localize "gray" failures in datacenter networks, using end-to-end information, achieving as high accuracy as possible within roughly a few seconds (including monitoring and inference), making best use of available information, including active probes or INT (where paths are known) and passive monitoring (exact paths are unknown).

3.3 FLOCK DESIGN

We present Flock in three components. First (§ 3.3.1), Flock monitors end-to-end flows at the endpoints (e.g. hosts) of the network. The monitored flow observations, comprised of metrics from active probes and (when available) passive flow monitoring and/or INT, are then sent to a central collector. Next, the collector periodically constructs a probabilistic graphical model (PGM) from the flow observations (§ 3.3.2). This PGM captures uncertainty in how the observations may have resulted from underlying network components. Finally, Flock periodically performs MLE inference (§ 3.3.3) on the PGM model, searching through the exponentially-large hypothesis space (i.e., sets of possible faulty components) to find the hypothesis that maximizes the probability of the observed flow metrics according to the model. Our key contribution lies in the MLE inference algorithm, comprised of multiple optimizations to make it scale. Nevertheless, we describe all components for completeness.

3.3.1 Flow monitoring

Monitoring of end-to-end flows may occur in an agent process running on hosts, in the hypervisor of a virtualized data center, or potentially at edge/top-of-rack switches (e.g., with INT). Our inference algorithms are agnostic to where exactly these measurements come from. Still, for concreteness, we describe the operation of an endpoint monitoring agent (§3.5).

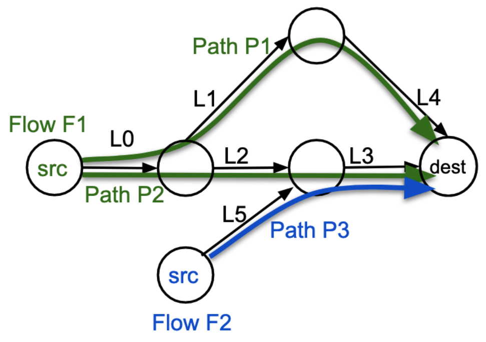
The agent periodically actively probes the network and may optionally passively observe performance of ongoing flows. Note that 007 also observes ongoing traffic, to decide what active probes to launch. Metrics from both active and passive monitoring are aggregated by flow, and optionally randomly sampled to reduce volume. Periodically, the agent sends these reports to the collector. For the rest of this section, we assume flow reports include RTT, source, destination, retransmissions, packets sent and the path if known via active probing/INT, else the set of possible paths.

3.3.2 Inference graph model

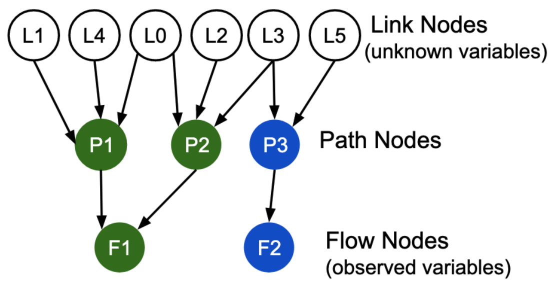
After telemetry is collected, Flock builds a probabilistic model of the network using two inputs: (1) telemetry reports as described above, potentially including active probes, INT, or passive telemetry, and (2) the network topology and routes. The latter could be provided by an SDN controller or a topology discovery tool and is used to determine a *set* of paths (typically, ECMP paths) that each flow may have traversed in the specific case of passive data with unknown paths.

Flock employs a probability model based on a Bayesian network that utilizes flow level metrics. A Bayesian network is a probabilistic graphical model that defines the probability distribution of a set of observed variables in terms of a set of unknown (or *hidden*) variables. It consists of a directed acyclic graph (DAG) where each node represents a random variable, unknown or observed, whose distribution can be specified as a function of its immediate ancestors in the DAG. The goal of the inference is to estimate the unknown variables (failed/not failed status of each component) given the observed variables (retransmissions, RTT, packets sent etc. associated with each flow).

We use a 3-layer graph model. Fig. 3.1 shows an example network with two flows and its corresponding model. Formally, the conversion from datacenter network to model is as follows. The **top layer** nodes in the graphical model represent individual links in the datacenter, referred to as *link-nodes*. Each link-node is a hidden binary variable which is 0 if



(a) Network



(b) PGM graph

Figure 3.1: (a) A network with flows F1 and F2 traversing several possible paths; (b) the corresponding PGM model.

the link has failed and 1 otherwise. The **intermediate layer** consists of nodes corresponding to paths in the datacenter, referred to as *path-nodes*. For each link ℓ in path p, there is an edge from the link-node of ℓ to the path-node of p in the Bayesian graph. A path-node represents an intermediate binary variable which is 0 if the path consists of a faulty link and 1 otherwise. Finally, the **bottom layer** consists of *flow-nodes* – one for every flow – which are the observed variables. A flow-node has an edge from each path-node in the path-set of that flow. We define the value of the flow-node variable as the number of *bad* packets – packets which experienced a problem. We describe two ways of setting the bad packets variable:

- Per packet analysis: For packet loss and corruption, we set the number of *bad* packets as the number of retransmissions which serves as a proxy for lost packets.
- Per flow analysis: To capture symptoms of high latency, we use a "per-flow" analysis which is in effect a special case of the per-packet model where the number of packets sent is 1, and the number of bad packets is set to 1 if the flow's RTT is above a threshold and 0 otherwise.

Flock's model assumes a flow F is routed via ECMP; F takes one of w paths chosen uniformly at random, and packets experience problems independently and uniformly at random. Thus, the probability of observing r bad packets out of the t sent can be given as:

$$P[F = (r,t)] = \frac{1}{w} \sum_{i=1}^{w} (1-\gamma_i) p_b^r (1-p_b)^{t-r} + \gamma_i p_g^r (1-p_g)^{t-r} \tag{3.1}$$

where γ_i is the value of the i^{th} possible path of F ($\gamma_i = 0$ if a failed link is on the i^{th} possible path and 1 otherwise). p_g and p_b are model hyperparameters: p_b represents the probability of a packet experiencing problems when taking a bad path (i.e., with at least one faulty link), and p_g represents the probability of a packet experiencing problems on a good path (no faulty links). Intuitively, a packet going through a failed link is much more likely to experience a problem, hence $p_b >> p_g$. § 3.5.2 describes how to pick p_g and p_b. Equation 3.1 can be adapted for weights (e.g. for WCMP [102]).

A *hypothesis* is an assignment $H \in \{0,1\}^n$ for all link-nodes. Equivalently, we can think of H as a *set of links* that are deemed to be failed, with all other link-nodes being not failed. The goal of the inference is to recover the hypothesis that consists of all truly failed links and only those links. Conditioned on a hypothesis H, the probability of the set of flow observations taking on the observed assignment of values (a certain number of bad packets r_i out of t_i packets sent, for each flow i) is simply the product of probabilities of all individual flow probabilities:

$$P[F_1, F_2, \ldots, F_m|H] = \prod_{F_i \in \text{flows}} P[F_i = (r_i, t_i)|H] = \prod_{F_i \in \text{flows}} P[F_i|H] \tag{3.2}$$

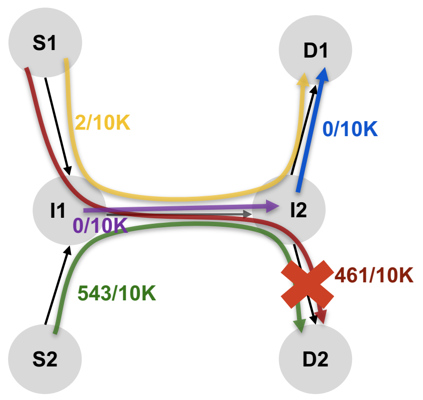
where F_i is shorthand for the event that flow i takes on the observed metric values, i.e., $F_i = (r_i, t_i)$.

Incorporating Priors. We assign a prior belief about failures by assuming that, a priori, any link can fail with probability ρ. The priors reduce the false positive rate by effectively assigning a lower prior to hypotheses with more links, thus favouring hypotheses with fewer failed links. If hypothesis H contains $|H|$ candidate failed links and there are n total links, then the likelihood of H after incorporating priors is given as:

$$Prior * \prod_{F_i \in \text{flows}} P[F_i|H]; \quad Prior = \rho^{|H|}(1-\rho)^{n-|H|} \tag{3.3}$$

Model extensions. The top layer nodes can include other component types besides links; we add device nodes, treating a device as another component in a flow's path, exactly as links. We found that a device prior that is 5× larger on log-scale, worked well in practice, as it forces Flock to detect a device failure only when there is stronger evidence for it than a link failure. Other components (line cards, racks, pods etc.) can be modeled in a similar way, but is beyond the scope of this paper.

Model Intuition. The model effectively incorporates several kinds of uncertainty. Given an observation of bad packets in a flow, we may not know what path is responsible (modeled via flow nodes having multiple path parents). Even if we do know the path, as for active



(a) Example: failed link shown via the red cross

Scheme	Predicted failed links
007	(I1, I2)
NetBouncer	(S2, I1), (I2, D2)
Flock	(I2, D2)

(b) Output of various schemes

Figure 3.2: Example: (a) 5 links in the network. 5 flows shown in 5 different colors, annotated with packets dropped/packets sent. (b) Flock correctly localizes the failed link.

probes and INT, we don't know what link is responsible[16] (modeled via path nodes having multiple link parents). Even if we know what link is responsible, an observed bad packet might or might not mean there is a faulty link (modeled via both good and bad links having non-zero probability of bad packets).

Differences from Sherlock's PGM. Sherlock was intended to model application-level failures, and thus includes elements that we don't need such as services and load balancers. Sherlock uses three node states – working, failed, and partially working; we omit the last, as Flock models some packet loss even for working links. Starting from Sherlock's model and making these changes results in a PGM that is very close to Flock's, except for the probability formulations.

Model Assumptions. Like any other model [25, 62, 97], ours has assumptions: fixed packet fail probabilities (as in [62]), classifying paths as failed on just the absence/presence of at least one failed link (as in [97]), and packets getting affected independently (as in [97, 62, 25]). We tried different assumptions – one with variable fail rates obtaining MLE using Nesterov's Gradient Descent algorithm, but it was too slow; another variation that treats paths differently depending on their number of failed links, but its accuracy was worse.[17] We present the model that gave the best results in experiments based on real environments that don't adhere to any model assumptions. We also give theoretical (§ 3.4.2) evidence

[16]While INT can capture per-hop metrics, it may not detect where a gray failure happens. For example, a silent corruption of packet data likely would not be detected until the packet reaches the receiver's host stack.

[17]Note that Flock still localizes multiple failures on a path since there are flows that transit one failed link, but not the other.

supporting Flock's effectiveness with these assumptions. Finally, it's important to keep in mind that the model does not need to match reality perfectly, *it only needs to be "close enough" that the most likely explanation in the model is the right one.* Fig: 3.2 illustrates how the PGM-inference can localize more accurately than past schemes.

3.3.3 Inference algorithm

We describe the inference algorithm next. For ease of exposition, this description only considers link failures. Devices are treated identically to links and our implementation handles both.

Recall that a hypothesis H is a candidate set of failed links. From the model, we can compute the probability of the flow variables taking their observed values (number of bad/sent packets for each flow) given H; this is the likelihood of H. We denote this likelihood as $L(H) = P[F_1|H]P[F_2|H]\dots P[F_m|H]$. The maximum likelihood estimator $\mathcal{H}$ is the hypothesis that maximizes $L(H)$, or equivalently log likelihood $LL(H) = \log L(H)$, that is $\mathcal{H} = \underset{H \subseteq \text{links}}{\arg\max}\ LL(H)$.

We normalize all likelihoods by the likelihood of the no-failure hypothesis (i.e., $H_0 = \{\}$) to cancel out any flow whose path set does not include any failed links in a hypothesis H.

The goal of MLE inference is to compute $\mathcal{H}$. Simply computing the likelihood of each possible hypothesis would be impractically slow because there are 2^n hypotheses, where n is the number of links. Sherlock [25] and NetSonar [97] limit max concurrent failures to k, reducing the search space to $O(n^k)$ hypotheses. However, in our setting, this is still far too slow, even for $k = 2$ (§ 3.7.8). Further, a datacenter can have many concurrent failures making $k > 3$ important.

We introduce two algorithmic techniques to accelerate MLE inference in PGMs. *Greedy search* reduces the number of hypotheses examined. While Greedy MLE is a simple idea, our main contribution is showing that it finds good solutions even after narrowing the hypothesis space: we provide theory (§ 3.4.2) and experiments (§ 3.6). *Joint likelihood exploration* (JLE) is a new algorithmic technique, that reduces the time per examined hypothesis. Both techniques provide significant speedups individually and together speedup the inference by several orders of magnitude (§ 3.7.8).

Greedy Search: We start from the no-failure hypothesis and extend it one link at a time. Specifically, we maintain a current hypothesis H. Initially, $H = \{\}$. In each iterative step, we scan over each link $l \notin H$ and calculate $LL(H \cup \{l\})$. If one of these log likelihoods improves over $LL(H)$, we set $H := H \cup \{l^*\}$ where l^* is the link offering the biggest improvement, and continue iterating. When no added link failure improves the log likelihood of the current hypothesis H, the search terminates and returns $H_{greedy} = H$.

There is still a performance challenge with Greedy MLE. Each iterative step requires evaluating close to n hypotheses (specifically $n - |H|$) to find l^*. Even with 40 cores, greedy search took over 3 hours for a medium-sized datacenter (§ 3.7.8). This motivates our second key optimization.

Joint likelihood exploration (JLE): We devise an additional acceleration technique for inference algorithms for PGMs with discrete variables. We use it to speed up each iteration of the greedy algorithm by a $O(n)$ factor, where n is the number of components (links and switches). Note that greedy+JLE produces the exact same solutions as greedy.

Suppose we are given the current best hypothesis H which has the maximum likelihood among hypotheses searched till now. Joint likelihood exploration is a technique to quickly explore all "neighbors" of H – all assignments that are different from H in the inclusion or exclusion of exactly one link. Note that there are n such neighbor hypotheses of H.

Definition 3.1. *Let $H \oplus l$ denote the hypothesis obtained by flipping the status of link l in H, i.e., if $l \in H$ then $H \oplus l = H \setminus \{l\}$ and otherwise $H \oplus l = H \cup \{l\}$. Let $\Delta_H(l) = LL(H \oplus l) - LL(H)$ represent the difference in log likelihoods of hypotheses $(H \oplus l)$ and H.*

JLE Intuition: We explain JLE by showing how it accelerates the Greedy algorithm (although it can also accelerate exhaustive search). In each iteration, Greedy computes each $LL(H \cup \{l\})$ for each l, to find the link l^* offering the most improvement. Note that maximizing $LL(H \cup \{l\})$ over all l is equivalent to maximizing $\Delta_H(l)$ over all l. So, in each iteration of Greedy, we could compute an n-element array Δ_H whose elements are the values $\Delta_H(l)$ for each link l, and then scan the array to find the largest value. This finds the same l^* as in the original greedy algorithm.

But how do we compute the array Δ_H? If we do it in the obvious way, by iterating over each l and directly computing $LL(H \oplus l) - LL(H)$, then this is nearly identical to the original greedy algorithm which computed $LL(H \cup l)$, with no appreciable change in runtime. What JLE offers is *a faster way to compute* Δ_H, with careful algorithm engineering. This involves two somewhat different types of computation: quickly preparing the array Δ_{H_0} for the first iteration; and quickly iteratively *updating* the array for each subsequent iteration.

To initially create Δ_{H_0}, note each entry $\Delta_{H_0}(l)$ represents the difference in log likelihood due to failing link l, compared to no failures. To compute these differences, we only have to look at the effect on flows whose paths intersect l. Furthermore, there is an important opportunity for memoization in computing the log likelihood-difference formula across different links l: the effect on a flow's likelihood depends only on the number of failed paths, not the specific failed links.

Now suppose we have an existing Δ_H and the search algorithm is about to move from hypothesis H to hypothesis H' in its next iteration. We need to compute the array $\Delta_{H'}$. To do this, we track the *difference in the difference arrays* (Δ_H vs. $\Delta_{H'}$) rather than directly computing the difference in likelihoods of H and $H \oplus l$ for every l. The key insight here is that each entry of the difference array Δ_H can be written as a sum of contributions for all flows and only some of the terms need to be updated after moving to a new hypothesis H'. The fact that we can do this faster than creating the array from scratch is the key to JLE's acceleration.

JLE formalization: First the algorithm needs to compute the array Δ_{H_0} for the first iteration of Greedy. For details, we refer the reader to the pseudocode of function ComputeInitalDelta in Algorithm 3.3 in § 3.9, which follows the intuition above.

Next, we describe how the Δ_H array can be updated when the greedy algorithm moves to a new hypothesis H'. We first note that $LL(H)$ is a sum of contributions from all flows and can be written as $LL(H) = \sum_{F \in \text{flows}} LL_F(H)$, where $LL_F(H) = \log P[F|H]$. We have

$$\Delta_H(l) = LL(H \oplus l) - LL(H) = \sum_{F \in \text{flows}} LL_F(H \oplus l) - \sum_{F \in \text{flows}} LL_F(H) = \sum_{F \in \text{flows}} \Delta_H(l, F) \tag{3.4}$$

where $\Delta_H(l, F) = LL_F(H \oplus l) - LL_F(H)$. Next we derive a useful property about the individual flow contributions $\Delta_H(l, F)$.

Definition 3.2. *A flow F intersects with link l if at least one of the possible paths for F has link l.*

Theorem 3.1. *For a link l' and hypothesis H, let $H' = H \oplus l'$. Then for all links l and flows F,*

(i) If F does not intersect with l, then $\Delta_H(l, F) = 0$

(ii) If F does not intersect with l', then $\Delta_{H'}(l, F) = \Delta_H(l, F)$.

Proof. This can be easily seen by expanding $\Delta'_H(l, F)$ and $\Delta_H(l, F)$. We note that for any H, the log likelihood of a flow F, given by $LL_F(H) = \log P[F|H]$, does not depend on a link l if F does not intersect with l (that is, $LL_F(H) = LL_F(H \oplus l)$).

For (i), if link l does not intersect with F, then flipping l's status does not affect F: $LL_F(H) = LL_F(H \oplus l) \Rightarrow \Delta_H(l, F) = 0$.

For (ii), when l' does not intersect with flow F, $LL_F(H') = LL_F(H \oplus l') = LL_F(H)$ and $LL_F(H' \oplus l) = LL_F(H \oplus l' \oplus l) = LL_F(H \oplus l)$. Consequently, we get $\Delta_{H'}(l, F) = \Delta_H(l, F)$. □

Hence, to obtain $\Delta_{H'}(l)$ from $\Delta_H(l)$, we only need to update the terms $\Delta_H(l, F)$ for flows F that intersect with *both* links l' and l since all other flow contributions to $\Delta_{H'}(l)$ remain unchanged from $\Delta_H(l)$. Let flows(l', l) denote the set of flows that intersect with both l and l'. After updating the current hypothesis from H to H', we can compute the new entry $\Delta_{H'}(l)$ for link l using Theorem 3.1:

$$\begin{aligned}
\Delta_{H'}(l) &= \sum_{F \in \text{flows}} \Delta_{H'}(l, F) && (3.5)\\
&= \sum_{F \in \text{flows}} \Delta_H(l, F) + \sum_{F \in \text{flows}(l', l)} \Delta_{H'}(l, F) - \Delta_H(l, F) && (3.6)\\
&= \Delta_H(l) + \sum_{F \in \text{flows}(l', l)} \Delta_{H'}(l, F) - \Delta_H(l, F) && (3.7)
\end{aligned}$$

Once we have equation 3.5, the algorithm to update the Δ array, for all n entries, is simple to state. After moving to $H' = H \oplus l'$, we iterate over all flows that intersect with l'. For each such flow F, let L_F be the set of links that intersect with F. For each $l \in L_F$, we update F's contribution to $\Delta_{H'}(l)$. With memoization, one can update all entries $\Delta_{H'}(l, F)$ for all $l \in L_F$ in a couple of passes over L_F, similar to how we initially computed Δ_{H_0}. The crux of the greedy+JLE algorithm is outlined in Algorithm 3.1 of § 3.9 and the full pseudo-code is given in Algorithm 3.3 of § 3.9.

Given $LL(H)$, an alternate approach is to compute $LL(H \oplus l)$ without JLE, individually for each l, as in [25, 97]. This requires updating the contribution of all flows that intersect with l since their likelihoods $LL_F(H \oplus l)$ would have changed after flipping the status of link l. Thus, the number of flows whose contributions need to be updated for computing just one entry $LL(H \oplus l)$ for a single l is the same as that for computing all n entries of the Δ array jointly with JLE. Thus, JLE results in in a $O(n)$ speedup. The reason for this large improvement is that JLE tracks the change in the Δ's (i.e., the difference in the differences: $LL_F(H \oplus l) - LL_F(H)$) across iterations which allows reuse of computation from the previous iteration.

Besides greedy search, JLE can apply to any algorithm which explores a hypothesis H's neighbors: $H \oplus l$ for all l. This includes brute force, Sherlock and NetSonar's inference, and MCMC techniques (e.g. Gibbs sampling). Using JLE, we were able to accelerate (i) Sherlock's inference (Alg. 3.2 in section 3.9), and (ii) Gibbs sampling for Flock, both by multiple orders of magnitude. We ended up using Greedy for Flock because (i) Sherlock's inference can not detect $K > 2$ concurrent failures and was still slow with JLE (§ 3.7.8) and (ii) for Gibbs sampling, it's hard to bound the number of iterations required for convergence. Gibbs sampling [77] without JLE was too slow for our purposes.

3.4 FLOCK: ANALYSIS

3.4.1 Runtime analysis

Let n be the number of links, m be the number of flows, T be an upper bound on the number of links that any flow intersects with, D be an upper bound on the number of flows that any link intersects with and K be the maximum number of concurrent failures (note that Flock's algorithm doesn't need to know K).

We describe the components in the running time of Flock's overall inference, including Greedy and JLE:

- Before the first greedy iteration, the Δ array is computed once, in a linear pass over all flows and their path sets, incurring $O(n + mT)$ time.
- After each greedy iteration, updating the Δ array via JLE requires iterating over all flows that intersect with the newly added link. For each such flow F, let L_F be the links that F intersects with. Updating all entries $\Delta_{H'}(l, F)$ for all $l \in L_F$ requires a couple of passes over L_F. Thus, the execution time for subsequent $(K - 1)$ greedy iterations, barring the first, is $O((K - 1)DT)$.

Hence, the running time of Greedy inference with JLE is $O(n + mT + (K-1)DT)$. If we had used just Greedy without JLE (computing likelihood of each hypothesis individually), the runtime would be $O(n + mT + (K-1)nDT)$.

We compare runtime with Sherlock. To compute the MLE, Sherlock scans all $O(n^K)$ hypotheses with $\leq K$ failures. Sherlock is better than brute force, however: it uses $LL(H)$, for an explored hypothesis H, to compute $LL(H \oplus l)$ by updating the flow contributions $LL_F(H)$ for all flows F that intersect with link l (since their likelihoods would have changed after flipping the status of link l), giving $O(n^K DT)$ runtime. We can apply JLE to accelerate Sherlock's inference by evaluating n neighbor hypotheses at once (Algorithm 3.2 in § 3.9), improving Sherlock's runtime, by a factor of n, to $O(n^{K-1}DT)$. However, this is exponential in K and too slow for our purposes. From the analysis above (and our experiments later), it can be seen that Greedy + JLE is dramatically faster.

3.4.2 Accuracy analysis

We analyze conditions in which Greedy returns the true MLE hypothesis. To make the problem tractable, we restrict the analysis to cases where path taken is known (true for active probes and INT) and packets crossing a link get dropped independently according to a (unknown) drop probability of that link (inference is NP hard if packets get dropped adversarially, see Theorem 3.3 in 3.10. Theorem 3.2 below gives a sufficient condition on the traffic for Flock's inference to correctly recover the set of failed links, providing intuition for why Flock works well in practice.

Definition 3.3. *For given traffic T, let $T(\{l_1, l_2, ...l_k\})$ denote the number of packets that each go through all of the links $\{l_1, ...l_k\}$. If $T(\{l_1, l_2\})/T(\{l_1\}) \leq \epsilon$ for all links l_1 and l_2, we say T is ϵ-skewed.*

Theorem 3.2. *For any topology with $(1/\alpha)$-skewed traffic, with high probability, Flock's inference returns the set of all failed links if the number of failures is $\leq \alpha/2$, the number of packets T_{min} crossing every link is larger than a certain threshold, and the drop probabilities are $< p_g$ on all good links and $> p_b$ on all failed links where (p_g, p_b) satisfy the condition $5p_g < p_b < 0.05$.*

See § 3.10 for proof. Theorem 3.2 has an intuitive interpretation: Consider two links l_1 and l_2, where l_1 has failed and l_2 works correctly. Intuitively, at most $\frac{1}{\alpha}$ fraction of the dropped packets on l_1 will transit l_2. One can think of this as l_2 getting "$\frac{1}{\alpha}$ fraction of the blame" for the dropped packets on l_1. If there are $(f\alpha)$ failures and these are arranged adversarially so that all of them add blame to l_2, then in the worst case l_2 can get $(f\alpha) \cdot \frac{1}{\alpha} = f$ times as much blame as a true failed link. Intuitively, a large enough constant f would confuse the algorithm into classifying l_2 as failed. The theorem proves that for $f < 0.5$, the algorithm outputs the true failed links.

3.5 IMPLEMENTATION

3.5.1 Agent and inference engine

We implement an agent that runs on end-hosts and collects flow statistics via a lightweight packet dumping tool based on the PF_RING module [16] and periodically sends it to a collector in IPFIX format. Commercial solutions also exist [18, 17], possibly employing alternate approaches such as pulling flow statistics from the kernel via eBPF or using multiple collectors at scale. As the specifics of the agent/collector don't affect our results, we leave more optimized designs for future work.

Flock's inference engine, written in C++, (i) collects IPFIX flow reports from agents and (ii) periodically runs inference on the collected input. Currently, the network topology is static, but the inference engine can be modified to obtain the topology from a controller. The engine periodically reads flow reports from the queue, every 30 seconds, and runs the inference algorithm of § 3.3.3.

3.5.2 Parameter Calibration

An important problem we encountered, with all schemes, is how to set their hyperparameters. Flock has 3 hyperparameters (p_g, p_b, ρ), NetBouncer has 3, and 007 has 1. In real deployed systems, parameters and thresholds are quite common, and are set based on past deployment experience. However, manual tuning puts extra onus on the user and makes it difficult to evaluate systems. The manually set parameters of past schemes 007 [24] and NetBouncer [90] gave suboptimal results in our environments, across different topology scales and failure scenarios.

Thus, we design an automated parameter calibration method that we use for all schemes. We use a training set of monitoring data to search for the parameter settings that obtain the best precision and recall in the training set. It is tempting to obtain the training set from historical monitoring data, which is generally available in deployed systems. But this can be tricky, since: (a) historical data may not have faults labeled; and (b) faults, especially rare faults, may be diverse and unpredictable so that historical data is not entirely representative of the next upcoming incident.

To solve (a), we leverage simulations to obtain the training set. We calibrate parameters once using this training set and use those parameters across our experiments, unless stated otherwise. Note that if this method were used in a deployment, it would increase concerns with (b) since simulations may not match the real world. To address (b) we will experimentally quantify the robustness of each scheme to scenarios where the train and test sets are drawn from *different environments* (different topology, monitoring duration, fault rate, fault type).

Once we have the training set, we use the following calibration method. For each hyperparameter, we choose equally-spaced values in a reasonable range of possible values. We fix a minimum precision P and find the parameters which, in a training set, yielded highest recall and had precision $> P$. Varying P produces a set of parameters that are Pareto-optimal along the precision/recall tradeoff curve. Our evaluation will apply these parameters to a separate test data set.

To choose a single parameter setting (rather than a tradeoff curve), we set $P = 98\%$ and find the setting that maximizes recall (in the training set); if no such point exists or recall is too low ($< 25\%$), then we subtract 5% from P and try again, repeating until a setting is found. This method lays more emphasis on precision, which is usually desirable.

3.6 EVALUATION METHODOLOGY

3.6.1 Systems evaluated

We compare Flock with several state-of-the-art datacenter fault localization schemes. Our codebase consists of 7K LOC including C++ implementations of all inference algorithms.

We implemented the "Ferret" inference algorithm of **Sherlock** (Sec. 3.2 of [25]). For a fair comparison, we run Ferret on the same PGM as Flock (which is anyway similar to Sherlock's; § 3.3.2). Sherlock can not detect $K > 2$ failures, but (as expected) resulted in the same accuracy as Flock for $K \leq 2$ failures at small scale. Hence, we only show performance differences. We implemented **NetBouncer's** algorithm (Fig. 5 in [90]) and **007** (Algorithm 1 in [24]). We verified our 007 implementation by matching it with the publicly available 007 code and reproducing Fig. 10 in [24].

For all schemes, we calibrate parameters once using simulations of random packet drops and use those parameters by default, unless stated otherwise.

3.6.2 Input telemetry types

We use four different kinds of input for inference:

- **A1**: Active probes between end-hosts and the core switches with known paths, as designed for NetBouncer [90].
- **A2**: Reports about flows with ≥ 1 retransmission, along with their (actively-probed) paths, as designed for 007 [24].
- **P**: Passive information consisting of reports about regular (application) data flows, whose traffic matrix is thus dictated by the network environment (§ 3.6.3). A set of possible paths is known (based on ECMP multipath).
- **INT**: We assume INT [14] provides reports, including paths, for both A1 and P (which thus becomes a superset of A2).

Note the last case is intended to test full deployment of INT; in other deployment modes it could trace just a subset of traffic. Similar reports could be obtained from other recent packet marking [83] or mirroring [104, 88, 51, 60, 89] methods. Since Flock can incorporate different kinds of inputs, we compare accuracy across input types: Flock (A1) vs NetBouncer (A1), Flock (INT) vs NetBouncer (INT) and Flock (A2) vs 007 (A2). We also quantify the accuracy boost Flock obtains from additional passive information (A1+P, A2+P, A1+A2+P). NetBouncer and 007 cannot trivially ingest the passive telemetry as they do not model path uncertainty. The passive flow telemetry can be downsampled in a large datacenter with high link speeds to reduce volume of the monitoring data.

3.6.3 Network environments

NS3 simulations. We set up a NS3 simulation to output a trace consisting of flow metrics (retransmissions/packets sent). We feed this trace as input to inference. We use a standard 3-tiered Clos topology [20] with 2500 40Gbps links, ECMP routing and 3x oversubscription at ToRs. Like [90], we set drop rates on all non-failed links between $0 - 0.01\%$ chosen independently and uniformly at random to model occasional drops on good links. For all our experiments, half the traces used uniform random traffic and the other half used a skewed traffic pattern where 50% of the traffic is concentrated among 5% of the racks, randomly chosen. Flow sizes were drawn from a Pareto distribution (mean: 200KB, scale:1.05) to mimic irregular flow sizes in a typical datacenter [21].

Large scale simulation. NS3 was too slow for large scale simulations. Hence, we use a flow level simulator (similar to [24]), that drops each packet as per preset drop probabilities on links but does not model queuing or TCP. We use this simulator for scaling experiments (§ 3.7.8).

Hardware test cluster. We set up a physical testbed with 10 switches and 48 emulated hosts, each with its own dedicated hardware NIC port and one CPU core. One of the 48 hosts runs Flock's collector. We use a standard 2-tier Clos topology with 2 spines, 8 leaf racks and 6 hosts per rack. We provision 1 Gbps link speeds to emulate as many hosts as possible. Schemes with A1 are omitted from our testbed results since our switches don't have the in network IP-in-IP feature for A1 [90].

3.6.4 Failure scenarios

In simulation:

- **Silent link packet drops:** a link drops a small fraction of packets without updating switch counters. Silent drops are a common problem in the industry [83, 24, 90].
- **Silent device failure:** An error in a device component (e.g., memory, line card) causes silent packet drops. This differs from the prior scenario since it affects many or all links on the device.

In hardware test cluster:

- **Queue misconfiguration**: A WRED queue [42] drops packets with probability p when the queue length is above a configurable threshold w. We misconfigure WRED queues on switches, setting $p = 1\%$ (choices: 1-100%) and $w = 0$ (so, the link works normally if the queue is empty).
- **Link flap**: We pull out a cable manually and quickly put it back in to emulate link flaps [36]. In our setup, link flaps caused the latency of the flows transiting the link to spike, but did not produce any significant increase in retransmissions (i.e., the link was buffering packets).

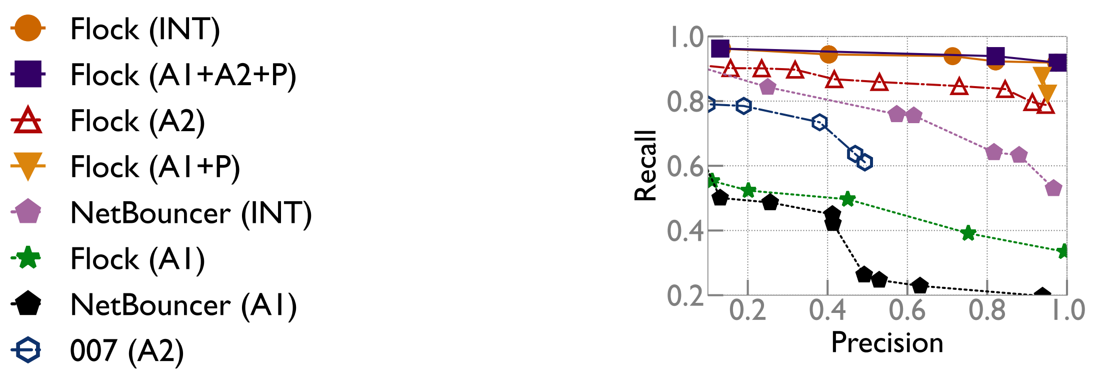
3.6.5 Evaluation Metrics

Precision is the fraction of predicted failed links that had actually failed and **recall** is the fraction of failed links that were correctly reported as failed. A faulty device or any of its links are considered to be correct for calculating precision. For calculating recall, including the faulty device itself in H counts as 100% recall, and including x% of the device links in H counts as x% recall, where H is the set of failed links/devices predicted by the algorithm. More precisely, if H is the set of failed links predicted by the algorithm and H^* is the actual set of failed links, then precision $=|H \cap H^*|/|H|$ and recall $= |H \cap H^*|/|H^*|$.

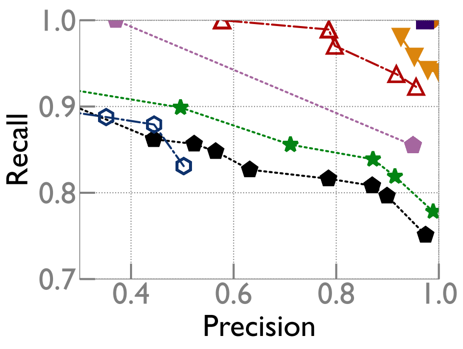
We define precision to be 1 if the algorithm returns the empty hypothesis. For 0 actual failures, precision represents the fraction of examples where the algorithm returns a wrong answer and recall is 1 since there are no failures to detect. We use the standard **Fscore** measure (harmonic mean of precision and recall) when we need a combined measure of accuracy.

3.7 EVALUATION RESULTS

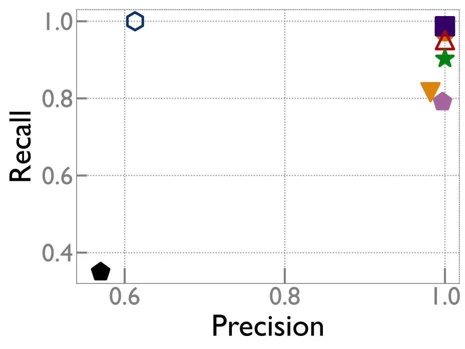
The first goal of our evaluation is to investigate Flock's accuracy compared to NetBouncer and 007. We compare various input types (INT, A1, A2, P) to quantify benefits of incor-



(a) With 100K flows

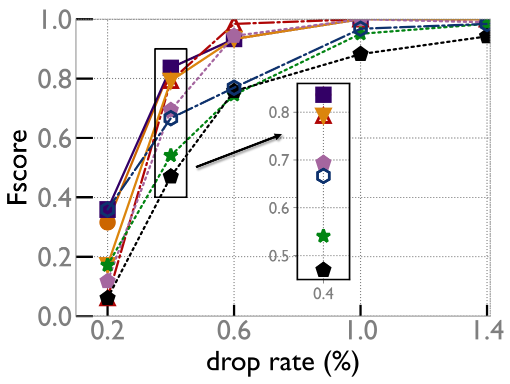


(b) With 400K flows

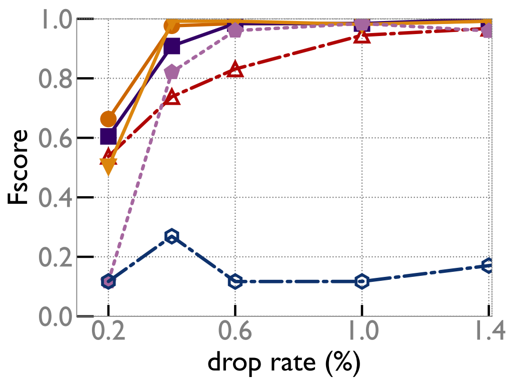


(c) Device failures

Figure 3.3: Accuracy for silent packet drops. (a), (b): Tradeoff curves for NetBouncer, 007 and Flock for silent drops, varying hyperparameters for each scheme. Schemes are annotated with the input data they use. (c): Accuracy on device failures.



(a) Uniform traffic



(b) Skewed traffic

Figure 3.4: The extent of drop rates that each scheme can detect (each point averaged over 32 traces).

porating passive data and INT. As expected, Flock and Sherlock had the same accuracy in small scale experiments (with max $K = 2$ failures), where Sherlock finished in reasonable time. Hence we don't show accuracy comparisons with Sherlock. Next, we investigate performance of Flock compared to Sherlock, NetBouncer and 007 and the benefits of JLE in speeding up inference.

3.7.1 Silent packet drops

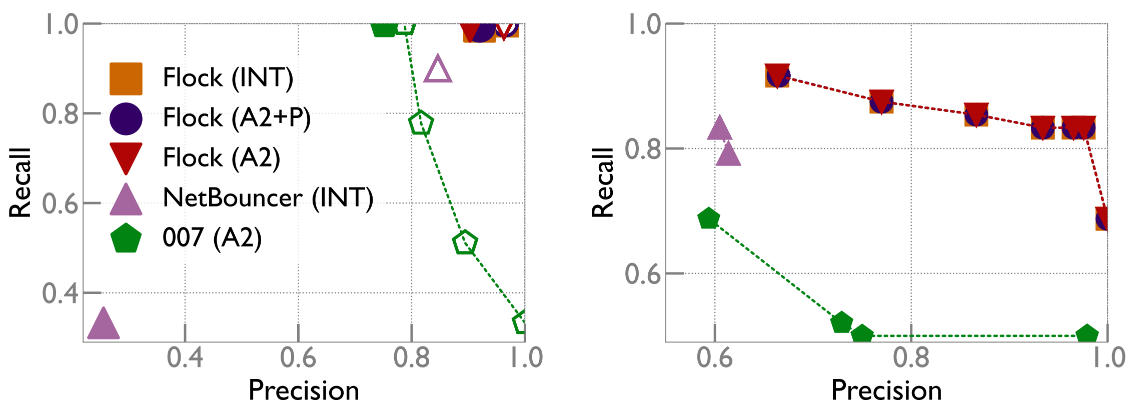
We generated 63 traces via NS3, each with 1 to 8 failed links with drop rate on each failed link chosen uniformly at random between 0.1% and 1% [90] (drops on good links are set as in § 3.6.3). We simulate active flows between the hosts and the core switches (A1), sending 40 packets/sec and 400K passive flows/sec across all hosts. Fig. 3.3 shows precision-recall tradeoff curves, with different parameters (§3.5.2) after 100K and 400K flows. With the chosen parameters for each scheme (§ 3.5.2)-

- **A1**: Flock reduces error rate over NetBouncer by roughly 45% (fscore: 0.5 vs 0.27). With $4\times$ more probes, Flock still reduces the error rate by >20% compared to NetBouncer (fscore: 0.87 vs 0.84).
- **A2**: Flock (fscore: 0.93) reduces error-rate over 007 (fscore: 0.61) by 5.5x after 400K flows.
- **A1+P and A1+A2+P**: When active probes (A1, A2) are augmented with passive information (P), Flock achieves very high accuracy (fscore with A1+A2+P: 0.98, A1+P: 0.93) after 400K flows (see Fig. 3.3a), suggesting that these schemes require less data than active-only schemes for localization.
- **INT**: Flock (INT) has the best accuracy (fscore 0.99) reducing error over NetBouncer (INT) (fscore 0.88) by 12x.

The last two results highlight the benefits of incorporating passive data for accuracy. 007's poor accuracy is likely due to its sensitivity to traffic skew (§ 3.7.3).

3.7.2 Device failures

Using the same setup as § 3.7.1, we simulate a device failure by failing f% of a faulty device's links. We generate 64 traces, each consisting of up to 2 device failures, varying f across traces from 25% to 100%. A subset of links failing on a device is similar to the behaviour of a faulty line card on that device. For all schemes, we used the same parameters as in § 3.7.1 (we calibrated NetBouncer's threshold for the number of problematic flows crossing a device).



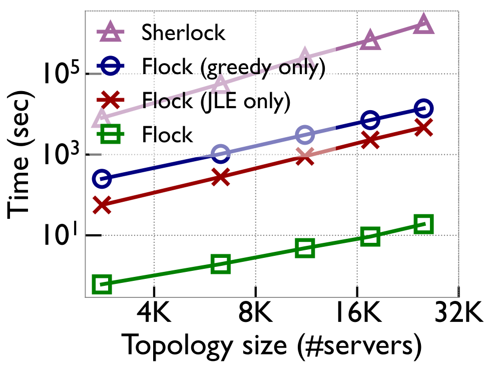
(a) Misconfigured queue (b) Link Flap

Figure 3.5: (a), (b) Accuracy on failure scenarios in testbed (solid markers). For (a), for comparison, we also show precision recall tradeoff curves with recalibrated parameters (hollow markers)

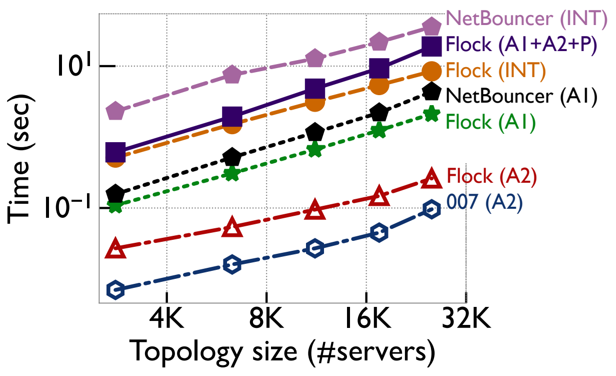
As shown in Fig. 3.3c, Flock outperforms NetBouncer and 007 for all types of information. Flock (INT) achieves ≈100% recall, compared to 80% recall of NetBouncer (INT). Flock (A2) reduces error-rate compared to 007 by 8x (fscore 0.97 vs 0.76). Flock (A1+P) has poorer precision for device failures than link failures.

3.7.3 Soft gray failures

We vary the drop rate on a single failed link to test what drop rates Flock can detect. A useful metric is the ratio of drop-rate on a failed link and the maximum drop-rate on a functioning link, which we call Signal to Noise Ratio (SNR), by a slight abuse of terminology. We use the same setup as § 3.7.2. From Figs. 3.4a, 3.4b, we conclude that Flock can detect links with $> 1\%$ drop rate (or SNR > 100) with high recall, with A2. With uniform traffic, 007's accuracy is good when SNR > 100 (consistent with the SNR in Fig. 10 of [24]). 007's recall gets affected significantly with skewed traffic (even with parameters calibrated separately for skewed traffic). After adding passive telemetry with either INT or (A1+A2+P), Flock's accuracy gets boosted and it is able to detect $> 0.4\%$ drop rate reliably. NetBouncer and Flock, both do well for A1, but NetBouncer's accuracy becomes worse for multiple concurrent failures with different drop rates (§3.7.1). Schemes for A1 (active probes) are unaffected by skew in the application traffic and hence omitted from Fig. 3.4b.

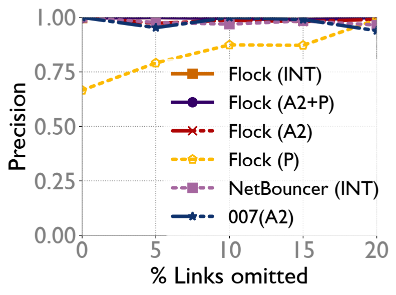


(a) Benefit of greedy and JLE

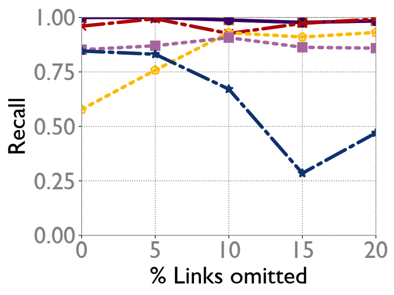


(b) Scheme runtime

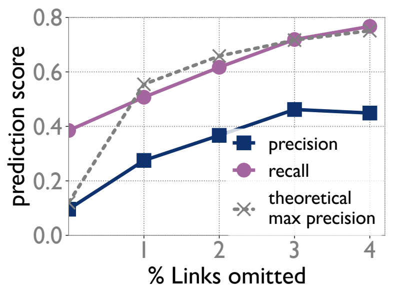
Figure 3.6: (a) Running time vs. a past PGM scheme (Sherlock); Flock achieves the same accuracy while being $> 10^4$x faster. Also shown is the effect of Flock's two optimizations (JLE, greedy) alone. (b) Running time of all schemes on various topology sizes.



(a) Precision



(b) Recall



(c) Flock(P) on a hard scenario

Figure 3.7: (a),(b) Accuracy on “irregular” Clos networks with a few links omitted. (c) Flock (P) produces useful results in a hard scenario, where other schemes don't apply. The “theoretical max preicision” is obtained from topology's equivalence classes.

3.7.4 Misconfigured queue

We move now from simulated faults to our testbed, beginning with misconfigured queues. Flock achieves high accuracy with all information types (see Fig. 3.5a). Using the same parameters as in § 3.7.1 for all schemes, Flock (INT) had higher precision and recall than NetBouncer (INT) (16x less error in fscore), whereas Flock (A2) had higher precision than 007 (A2) (the solid markers in Fig. 3.5a). For comparison, we also show precision recall tradeoff curves with parameters calibrated on the testbed with real examples. In this case, Flock (INT) had 7x less error than NetBouncer (INT) (fscore: 0.87 vs. 0.98) and Flock (A2) had 16x lower error compared to 007 (fscore: 0.97 vs. 0.5) (the hollow markers in Fig. 3.5a). Flock (A2+P) gets very close to Flock (INT), consistent with §3.7.1.

Parameters calibrated for → (D: different, S: same)		Different topology		Different failure rate		Different monitoring interval		Different failure scenario		Aggregate score (average Fscore)
p: precision, r: recall		p	r	p	r	p	r	p	r	
Flock (A1+A2+P)	D	0.92	0.98	0.97	1	0.98	1	0.95	0.99	0.973
	S	0.96	0.98	0.97	1	0.99	1	0.96	0.99	0.981
Flock (A2)	D	0.90	0.99	0.96	1	0.86	0.97	0.94	0.98	0.948
	S	0.94	0.98	1	1	0.94	0.92	0.94	0.98	0.962
Flock (INT)	D	0.92	0.99	0.96	1	0.98	1	0.95	0.99	0.973
	S	0.96	0.99	0.96	1	0.99	1	0.96	0.99	0.981
007 (A2)	D	0.75	1	0.87	1	0.51	0.82	1	0.33	0.728
	S	0.82	0.74	1	1	0.47	0.87	0.82	0.74	0.792
NetBouncer (INT)	D	0.25	0.33	0.95	1	1	0.66	0.28	0.33	0.589
	S	0.81	0.9	1	1	0.95	0.85	0.81	0.9	0.901

Table 3.1: Evaluating parameter robustness. For each scheme, we show accuracy when its parameters are calibrated on a different environment than the test data set (D) and when they are calibrated in the same environment (S).

3.7.5 Link flaps

We use a per-flow analysis (§ 3.3.2), classifying a flow as problematic if its RTT is > 10 ms. Since the analysis is per-flow and not per-packet, we had to recalibrate parameters. Flock does not model acks traversing the reverse path, which is important in this case. Accounting for that with a revised model would be possible, we leave that for future work. Even with a somewhat inaccurate model, Flock (INT) reduces the error rate by 1.66x over NetBouncer (INT) (fscore: 0.81 vs 0.69) and Flock (A2) reduces the error rate over 007 by 1.8x (see Fig. 3.5b).

3.7.6 Irregular Clos topologies

Real world datacenters are rarely perfectly symmetric like a Clos topology and typically have asymmetries due to failures, policies, piecemeal upgrades, etc. To see the effect of topology irregularity, we omit links from the fat tree. We recalibrated parameters on the irregular topologies for all schemes since the topology can be known in advance. (we also tested without recalibration; Flock's improvement over other schemes was even higher in this case. We omit the results for brevity.)

Fig. 3.7a, 3.7b show that Flock's accuracy is robust to topology irregularity. 007 is sensitive to topology irregularity, probably because its effect is similar as having traffic skew. We omit A1 since its active probing mechanism is designed only for regular Clos topologies.

Flock with passive only input (P): Some networks may only have passive information available. Past fault localization schemes can not be applied to passive only input since they don't handle path uncertainty. Operators in this situation resort to manual troubleshooting (traceroutes, adjustments to routing or taking links offline, etc.) which can take days. Flock with only passive input (P) can provide partial analysis (Fig. 3.7a, 3.7b). Interestingly, Flock (P)'s accuracy actually *improves* with more links removed. This is because in a symmetric Clos topology, there are equivalence classes of links (e.g. all links from a leaf switch to the spine layer) that cannot be differentiated because they participate in the same ECMP paths. As the topology becomes irregular, this breaks symmetry, and Flock's inference algorithm automatically takes advantage of this.

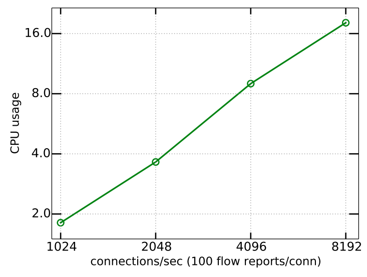
Fig. 3.7c shows an even more difficult fully passive scenario where the failed link is one of several symmetric links in a Clos topology, which has little irregularity for Flock to leverage and no active probes or path tracing. Flock (P) achieved >75% recall and >40% precision (thus, narrowing down the fault to about 2-3 possibilities). We believe this can be a very helpful starting point for operators.

3.7.7 Parameter calibration robustness

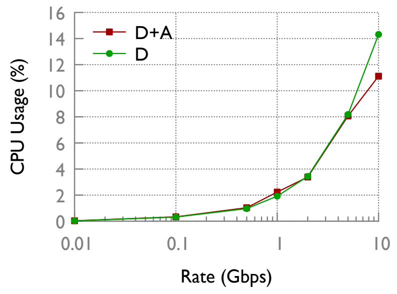
All schemes we consider have parameters that must be set (§ 3.5.2). We calibrated them via simulations with known ground truth. What happens when these systems encounter unexpected situations?

To test robustness, we trained each scheme on one environment and tested in a different environment. (This is effectively a strong form of cross-validation where not only are the train and test sets different, they are drawn from *different distributions*.) Specifically, we created different types of differences between train and test, changing the (a) duration of monitoring, (b) topology, (c) failure rate (training set has failed links with significantly different drop rates), and (d) failure type. In particular, for (b), the schemes were calibrated in our simulator with random packet drops, and tested on misconfigured queues in a 20x smaller topology in our physical testbed. Table 3.1 shows the accuracy in these cases, both when the train and test sets are drawn from different distributions ("D") and when they are drawn from the same distribution ("S"). The table's aggregate score column shows Flock was fairly robust to parameter calibration in a different environment, with under 2% loss in accuracy. 007 was also robust (6% loss) while NetBouncer was more sensitive (31% loss).

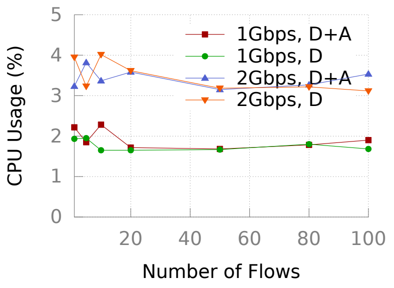
We also tested Flock's parameter sensitivity, i.e., how precision and recall vary with per-



(a) Collector scaling



(b) agent CPU usage: single flow



(c) several flows

Figure 3.8: (a) CPU usage at the collector, varying number of agents (b) CPU usage on the agent for handling a single flow. D: cost of packet header dumping, A: cost of compiling flow reports from packet headers.(c) agent CPU usage with many concurrent flows

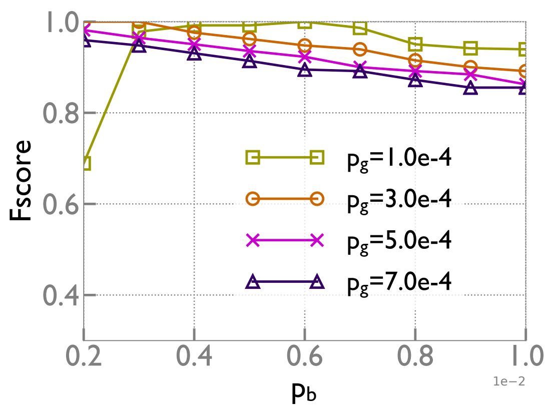
turbations of its parameters. Accuracy remained high for many choices of parameters (figure 3.9a).

3.7.8 Running time and scalability

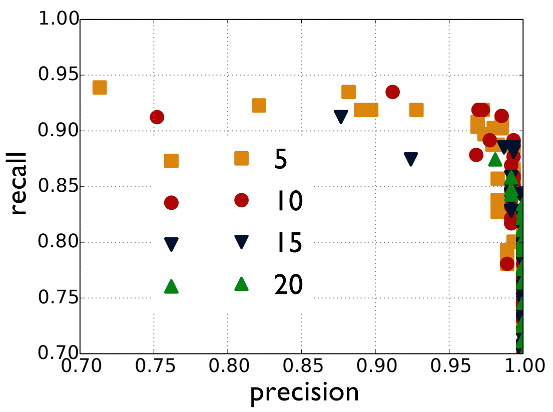
Algorithm runtime: Flock's main algorithmic innovation is its fast PGM inference compared to Sherlock's PGM inference. Fig. 3.6a shows Flock's inference is more than 4 orders of magnitude of faster than Sherlock, whose runtime on a large network was estimated to be 19 days, based on extrapolating a partial run. Recall Flock employs two optimizations: greedy and JLE. Fig. 3.6a shows each of these optimizations alone yields a $\approx$ 100x improvement over Sherlock.

Fig. 3.6b compares Flock to the non-PGM schemes. Flock is faster than NetBouncer on the same input data. 007 is the fastest but its time savings ($<$ 1 sec) does not trade-off well with accuracy.

Agent/Collector: Our agent's CPU usage for a end-host sending traffic grew linearly with the data rate (see Fig 3.8b) and was $<$ 2% of one core for a 1Gbps uplink and 10-15% of a core for a 10 Gbps uplink. As can be seen from figure 3.8c, the resource usage was independent of the number of flows. We verified that our multicore collector can handle 8K connections/sec from agents (see Fig 3.8). We tested this by launching several agent processes that generate dummy flow reports to send to the collector . These results show that the passive monitoring can be handled by an end-host agent and a centralized collector. As other commercial solutions also exist [17, 18], we leave more optimized agent/collector designs for future work.



(a) Parameter sensitivity



(b) Effect of changing priors

Figure 3.9: (a) Effect of changing p_g and p_b: intuitively, we expect precision to increase when p_g or p_b is increased, at the cost of reduced recall, which is confirmed by the figure (b) Higher priors result in points to the right. Priors ρ resulted in a significant reduction in false positives.

3.8 RELATED WORK

Many other works have studied fault localization in other contexts – for troubleshooting reachability, black holes in IP networks [66, 39, 48, 97], virtual disk failures [98], performance problems in distributed services [25, 74, 44] and application performance anomalies [96]. Some of these can benefit from PGM-based inference, accelerated via JLE. We leave this for future work.

Detecting entire flow drops, for e.g., caused by a misconfigured ACL or a forwarding loop, is challenging for end-to-end schemes (as noted in [90]), partly due to (un)available input when paths are fully blocked. Other schemes such as NetSeer [103] and Omnimon [60] or network verification [72, 64, 65, 43] are more suitable for this class of faults.

Several works orchestrate active probes for inference [67, 62, 71, 101, 33, 57, 90, 13, 50]. Flock can handle active probes and outperforms one such approach (NetBouncer). Additionally, it can use passive data for accuracy gains. deTector [79], MaxCoverage [66], Tomo [39] and Score [67] find a minimal set of components that explain most of the problems (e.g. packet drops). We expect them to run into similar problems as 007, since they don't account for traffic skew. Simon [47] reconstructs queuing times from active probes. This allows it to diagnose high latency, but not silent packet drops. Packet mirroring [104] can catch packet drops that happen in the switch pipeline (e.g. congestion drops), but can not detect silent interswitch or silent intercard drops. Flock is well suited to detect such problems. Pingmesh [50] and NetNorad [13] use active pings, but do not provide complete localization.

[83] identifies anomalies among symmetric links in a Clos network using statistical tests. It is sensitive to topology irregularity and requires path information.

3.9 PSEUDOCODE: JOINT LIKELIHOOD EXPLORATION AND GREEDY

Refer to Algorithm 3.1 for a short summary of Flock's inference algorithm and Algorithm 3.3 for full pseudocode.

Algorithm 3.1 Flock inference: Greedy search with JLE (crux)

```
 1: procedure GreedySearch()
 2:     current_hypothesis ← []
 3:     $\Delta$ = ComputeInitialDelta()
 4:     while max($\Delta$) > 0 do
 5:         link = argmax($\Delta$)
 6:         $\Delta$ = UpdateDeltaArr($\Delta$, current_hypothesis, link)
 7:         current_hypothesis.add(link)
 8:     return current_hypothesis
 9: procedure UpdateDeltaArr($\Delta$, hypothesis, link)
10:     new_hypothesis ← hypothesis + [link]
11:     for F in FlowsIntersectingWithLink(link) do
12:         for l in F.links do
13:             $\Delta$[l] += GetFlowDelta(new_hypothesis, l, F)
14:             $\Delta$[l] -= GetFlowDelta(hypothesis, l, F)
15:     return $\Delta$
```

Algorithm 3.2 JLE can be used to speedup Sherlock's inference, with max concurrent failures = K

```
 1: best_hypothesis = None
 2: max_likelihood = -$\infty$
 3: procedure SherlockWithJLE()
 4:     $\Delta$ = ComputeInitialDelta()
 5:     ExploreBranch([], $\Delta$, 0.0)
 6:     return best_hypothesis
 7: procedure ExploreBranch(current_hypothesis, $\Delta$, current_likelihood)
 8:     if current_likelihood > max_likelihood then
 9:         max_likelihood = current_likelihood
10:         best_hypothesis = current_hypothesis
11:     if current_hypothesis.size < K then
12:         for l in links do
13:             new_hypothesis = current_hypothesis.add(l)
14:             new_likelihood = current_likelihood + $\Delta$[l]
15:             $\Delta_{new}$ = UpdateDeltaArr($\Delta$, current_hypothesis, l)
16:             ExploreBranch(new_hypothesis, $\Delta_{new}$, new_likelihood)
```

Algorithm 3.3 Flock's Hypotheses search: Greedy with Joint Likelihood Exploration

```
1: procedure GreedySearch()
2:     current_hypothesis ← []
3:     $\Delta$ = ComputeInitialDelta()
4:     while max($\Delta$) > 0 do
5:         link = argmax($\Delta$)
6:         $\Delta$ = UpdateDeltaArr($\Delta$, current_hypothesis, link)
7:         current_hypothesis.add(link)
8:     return current_hypothesis
9:
10: procedure UpdateDeltaArr($\Delta$, hypothesis, link)
11:     new_hypothesis ← hypothesis + [link]
12:     for F in FlowsIntersectingWithLink(link) do
13:         ▷ these counters are a simple data structure trick to speed-up the subsequent for loop
14:         old_counters = GetCounters(hypothesis, flow)
15:         new_counters = GetCounters(new_hypothesis, flow)
16:         for l in F.links do
17:             $\Delta$[l] += GetFlowDelta(l, *new_counters)
18:             $\Delta$[l] -= GetFlowDelta(l, *old_counters)
19:     return $\Delta$
20:
21: procedure GetCounters(hypothesis, flow)
22:     paths_failed ← 0
23:     num_paths = dict()
24:     for path in flow.paths do
25:         if (PathFailedAsPerHypothesis(path, hypothesis)) then
26:             paths_failed++
27:         else
28:             for link in path do
29:                 num_paths[link]++
30:     return (paths_failed, num_paths[link], flow.paths.size(), flow.packets_sent, flow.bad_packets)
31:
32: procedure GetFlowDelta(link, paths_failed, num_paths, n_flow_paths, packets_sent,
    bad_packets)
33:     bad_paths = paths_failed + num_paths[link]
34:     return GetLogLikelihood(bad_paths, n_flow_paths, packets_sent, packets_dropped)
35:
36: procedure ComputeInitialDelta()
37:     for l in links do
38:         $\Delta$[l] ← 0
39:     for flow in flows do
40:         counters = GetCounters([], flow)
41:         for l in flow.links do
42:             $\Delta$[l] += GetFlowDelta(l, *counters)
43:     return $\Delta$
44:
45: procedure GetLogLikelihood(bad_paths, n_flow_paths, bad_packets, packets_sent)
46:     good_packets = packets_sent - bad_packets
47:     log_likelihood = bad_paths * pow($p_b$, bad_packets) *
48:                             pow(1 - $p_b$, good_packets)
49:     good_paths = n_flow_paths - bad_paths
50:     log_likelihood += good_paths * pow($p_g$, bad_packets)
51:                             * pow(1 - $p_g$, good_packets)
52:     log_likelihood /= n_flow_paths
53:     return log_likelihood
```

If the topology, flow statistics and path taken for each flow are chosen arbitrarily, then finding the MLE for such adversarial inputs, unsurprisingly, is NP-hard.

Definition 3.4. *Define adversarial inference as inference when the input is arbitrary, consisting of topology, flow metrics with arbitrarily chosen source and destinations, paths and arbitrarily chosen flow metrics (packets sent/dropped).*

However, we can make the following assumption, in the context of packet drops, to alleviate intractability for adversarial inputs: for each link, there is a (unknown) ground truth drop probability. Rather than adversarially, each packet crossing that link is dropped independently according to the drop probability of the link. First, we prove that adversarial inference is NP-hard.

Theorem 3.3. *Adversarial inference is NP-hard.*

Proof. We reduce the problem of finding a minimum vertex cover in a graph to the adversarial inference problem. For the proof, we will design an algorithm for min-vertex cover that makes polynomial number of accesses to an oracle O^{AI} for adversarial inference.

Let's say, we're given a graph G = (V, E) where V is the set of vertices and E is the set of edges and our goal is to find a minimum vertex cover in this graph. We create topology $\mathcal{T}$ for O^{AI} as follows-

- We create "vertex"-nodes n_v^1 and n_v^2 in $\mathcal{T}$ for each vertex $v \in V$ and attach them with a directed link l_v: $(n_v^1 \rightarrow n_v^2)$. Note that all vertex-nodes in $\mathcal{T}$ have exactly one link, either incoming or outgoing.

- For each edge $e \in E$, we create an "edge"-flow f_e, where f_e goes through links (l_{v_1} and l_{v_2}), where v_1 and v_2 are the endpoints of e. In order for this to be a legitimate path, we connect the endpoints of (l_{v_1} and l_{v_2}) to a special node.

- For each link l in $\mathcal{T}$, we create a "link"-flow f_l. going through just link l.

Finally we need to assign each flow, some number of packets sent and dropped. We first derive an expression for the return value of oracle O^{AI}. The output of O^{AI} is a binary assignment to links in $\mathcal{T}$ where we interpret a value of 0 as that link being failed and 1 as the link being up. Let $E_{\mathcal{T}}$ be the number of links in $\mathcal{T}$ and $\gamma_f^A = \prod_{l \in Path(f)} l$ denote the status of the path taken by flow a f with γ_f^A=0 being path failed and 1 being path not failed. Recall that a path is deemed to be failed if it contains at least one failed link. For flow f, if

n and r be the number of packets sent/dropped by the flow respectively, then the likelihood for flow f can be written as:

$$\begin{aligned}
\frac{P[f|H=\{l_1,l_2,\ldots l_{E_\mathcal{T}}\}]}{P[f|l_1=1,l_2=1,\ldots l_{E_\mathcal{T}}=1]} &= \frac{\gamma_f^A p_g^r(1-p_g)^{n-r}+(1-\gamma_f^A)p_b^r(1-p_b)^{n-r}}{p_g^r(1-p_g)^{n-r}} && (3.8)\\
&= \frac{p_b^r(1-p_b)^{n-r}}{p_g^r(1-p_g)^{n-r}}\left(1+\gamma_f^A\Big(\frac{p_g^r(1-p_g)^{n-r}}{p_b^r(1-p_b)^{n-r}}-1\Big)\right) && (3.9)\\
&= \frac{1+\alpha_f\gamma_f^A}{\alpha_f+1} && (3.10)\\
&= \frac{(1+\alpha_f\prod_{l\in Path(f)} l)}{\alpha_f+1} && (3.11)
\end{aligned}$$

where, $\alpha_f = \frac{p_g^r(1-p_g)^{n-r}}{p_b^r(1-p_b)^{n-r}} - 1 \in (-1,\infty)$ is a constant for flow f irrespective of H. The oracle O^{AI} then returns

$$\underset{H\in\{0,1\}^{E_\mathcal{T}}}{\arg\max}\prod_{f:\text{flows}} P[f|H] = \underset{H\in\{0,1\}^{E_\mathcal{T}}}{\arg\max}\prod_{f:\text{flows}}(1+\alpha_f\prod_{l\in Path(f)} l^H) \qquad (3.12)$$

Where l^H is the status (0/1) of link l as per hypothesis H. Given the above expression, we set the number of packets sent/dropped for all flows in the following way

1. For all edge-flows f_e, we set n, r: the number of packets sent/dropped such that $1+\alpha_{f_l} = \frac{1}{C}$ where $C >> 1$.

2. For all link-flows f_l where l is connected to a special node, we set n, r: the number of packets sent/dropped in such a way that if l is connected to a special node, then $1+\alpha_{f_l} = 1+C$. This ensures that O^{AI} will always assign a label of 1 to a link connected to a special node (recall that 1 denotes the link being up).

3. For all link-flows f_l where both endpoints of l are connected to vertex nodes, we set n, r: the number of packets sent/dropped such that $1+\alpha_{f_l} = 1+\epsilon$ where ϵ is a small number > 0. This assigns a small cost of assigning a label 0 to a link in $\mathcal{T}$ whose both endpoints of l are connected to vertex nodes.

The vertex cover is obtained by simply picking vertices in G corresponding to links in $\mathcal{T}$ that are deemed as failed by O^{AI}. Conditions (1) and (2) above ensure that only vertex-links will be classified as failed by O^{AI}. Conditions (2) and (3) ensure that the result set of vertices will cover all edges. Condition (3) ensures that the resultant vertex cover will be of the smallest size. QED.

Theorem 3.4. *For any topology with $(1/\alpha)$-skewed traffic, Flock's inference returns the set of all failed links if the number of failures is $\leq \alpha/2$ with high probability, the number of packets T_{min} crossing every link is larger than a certain threshold, and the drop probabilities are $< p_g$ on all good links and $> p_b$ on all failed links where (p_g, p_b) satisfy the condition $5p_g < p_b < 0.05$.*

The condition $T_{min} > T_0$ ensures that the effect of variance in the number of packets dropped by a link is small. Theorem 3.2 holds for a wide range of values of p_g and p_b (see lemma 3.1).

Proof. If γ_f^H be the 0/1 status of the path taken by flow f as per hypothesis H (0 being that the path has at least one failed link as per H and hence is labeled as failed), we can express the (normalized) log likelihood for a flow given a hypothesis as follows

$$\begin{aligned}
&\log P[f|H = \{l_1, l_2, \ldots\, l_n\}] - \log P[f|l_1 = 1, l_2 = 1, \ldots\, l_n = 1] && (3.13)\\
&= \log\left(\gamma_f^H p_g^r (1-p_g)^{n-r} + (1-\gamma_f^H) p_b^r (1-p_b)^{n-r}\right) - \log\left(p_g^r (1-p_g)^{n-r}\right) && (3.14)\\
&= (1-\gamma_f^H)\Big(r \log\frac{p_b}{p_g} + (n-r)\log\frac{(1-p_g)}{(1-p_b)}\Big) && (3.15)\\
&= (1-\gamma_f^H)\lambda(r - n\mu) = (1-\gamma_f^H)\lambda\sum_{i=0}^{n}(b_i - \mu) && (3.16)\\
&= (1-\gamma_f^H)X(f,H) && (3.17)
\end{aligned}$$

where $\lambda = \log\frac{p_b(1-p_g)}{p_g(1-p_b)}$ and $\mu = \log\frac{(1-p_g)}{(1-p_b)} \Big/ \log\frac{p_b(1-p_g)}{p_g(1-p_b)}$ are constants and b_i is a binary variable which is 1 if the i^{th} packet of the flow was dropped and 0 otherwise. One can check that $p_g < \mu < p_b$ for any $0 < p_g < p_b < 1$ by taking partial derivatives or using a plotting tool. Note that we can simply ignore the constant λ for the purpose of log likelihood maximization. If the maximum allowed drop rate on a correctly working link be p^* and the maximum path length for the given topology be k, then we set $p_g \geq kp^*$ so that for any flow f that does not go through any of the failed links , $(p_f - \mu) < 0$, where p_f is the packet drop probability of the path taken by f. We show the following lemma which holds for any reasonable settings for p_g and p_b for the purpose of detecting packet drops. The expected value of the $X(f,H)$ is given as

$$E[X(f,H)] = E\Big[\lambda \sum_{i=0}^{n} (b_i - \mu)\Big] \tag{3.18}$$

$$= \lambda \sum_{i=0}^{n} E[(b_i - \mu)] \tag{3.19}$$

$$= n\lambda(p_f - \mu) \tag{3.20}$$

Lemma 3.1. *If* $\mu = \frac{\log \frac{(1-p_g)}{(1-p_b)}}{\log \frac{p_b(1-p_g)}{p_g(1-p_b)}}$ *and* $5p_g < p_b \leq 0.05$*, then* $0 \leq p_g < \mu < 2\mu < p_b$

Lemma 3.1 can be seen by taking partial derivatives or via a numerical plotting tool.

Lemma 3.2. *If* p_l *denotes the drop probability of link* l*,* $L_f = \{l_1, l_2, ...l_k\}$ *be the links in the path taken by flow* f*,* p_f *denotes the drop probability of the path* L_f *and* H^* *denote the set of failed links, then* $(p_f - \mu) \leq \sum_{(l \in H^* \cap L_f)} p_l$

Proof. For any $l_i \in L_f$, we have-

$$p_f = 1 - \Big((1-p_{l_1})...(1-p_{l_i})...(1-p_{l_k})\Big) \tag{3.21}$$

$$= 1 - (1-p_{l_i})\Big((1-p_{l_1})...(1-p_{l_{i-1}})(1-p_{l_{i+1}})...(1-p_{l_k})\Big) \tag{3.22}$$

$$= 1 - \Big((1-p_{l_1})...(1-p_{l_{i-1}})(1-p_{l_{i+1}})...(1-p_{l_k})\Big) \tag{3.23}$$

$$+ p_{l_i}\Big((1-p_{l_1})...(1-p_{l_{i-1}})(1-p_{l_{i+1}})...(1-p_{l_k})\Big) \tag{3.24}$$

$$\leq 1 - \Big((1-p_{l_1})...(1-p_{l_{i-1}})(1-p_{l_{i+1}})...(1-p_{l_k})\Big) + p_l \tag{3.25}$$

Applying the same argument recursively for all links in $H^* \cap L_f$, we get:

$$p_f - \mu = 1 - (1-p_{l_1})(1-p_{l_2})...(1-p_{l_k}) - \mu \tag{3.26}$$

$$\leq \Big(1 - \prod_{l \in L_f \setminus H^*} (1-p_l)\Big) - \mu + \sum_{(l \in H^* \cap L_f)} p_l \tag{3.27}$$

$$\leq \sum_{(l \in H^* \cap L_f)} p_l \tag{3.28}$$

The last inequality follows from the fact that $(p_f - \mu) < 0$ for a path consisting of only good links. This completes the proof of lemma 3.2. QED.

Consider the set of hypotheses with single link failures- say S_1. We first show that $H_{opt} = \arg\max_{H \in S_1} L(H)$, corresponds to a failed link as $T_{min} \to \infty$ which in turns implies that the greedy algorithm succeeds in picking a failed link in the first iteration. Let $H_l \in S_1$ denote the hypothesis $\{l\}$ where l is a good link in the topology, $F(l)$ is the set of flows that go through the link l, n_f is the number of packets sent by flow f and p_f as before is the ground truth drop probability of the path taken by f. We have,

$$
\begin{aligned}
E[LL(H_l)] &= \sum_{f \in F(l)} E[X(f,H)] && (3.29)\\
&= \sum_{f \in F(l)} n_f(p_f - \mu) && (3.30)\\
&\leq \sum_{f \in F(l)} n_f \sum_{l^* \in H^* \cap L_f} p_{l^*} && (3.31)\\
&= \sum_{l^* \in H^*} \sum_{f \in F(l) \cap F(l^*)} n_f p_{l^*} && (3.32)\\
&= \sum_{l^* \in H^*} p_{l^*}\ T(l, l^*) && (3.33)\\
&\leq \sum_{l^* \in H^*} \frac{1}{\alpha}\ p_{l^*}\ T(l^*) \qquad \text{(since } \forall l^* \in H^*, l \neq l^*) && (3.34)\\
&\leq \sum_{l^* \in H^*} \frac{1}{\alpha} \sum_{F(l^*)} n_f p_{l^*} && (3.35)\\
&\leq \sum_{l^* \in H^*} \frac{1}{\alpha} \sum_{F(l^*)} n_f 2(p_{l^*} - \mu) \qquad \text{(since } 2\mu \leq p_{l^*}) && (3.36)\\
&< \frac{2}{\alpha} \sum_{l^* \in H^*} \sum_{F(l^*)} n_f(p_f - \mu) && (3.37)\\
&= \frac{2}{\alpha} \sum_{l^* \in H^*} E[LL(H_{l*})] && (3.38)\\
&\leq \arg\max_{l \in H^*} E[LL(H_l)] \qquad \text{(since } |H^*| \leq \alpha/2) && (3.39)
\end{aligned}
$$

This shows that the greedy inference algorithm will pick a failed link in the first iteration, say l_1. Greedily picking the link that maximizes the log likelihood of the hypothesis $\{l_1\}$ is equivalent to deleting all flows crossing l_1 and the link l_1 itself from the input for analysis for subsequent iterations. Hence, the same proof about iteratively picking a failed link works for subsequent iterations if we ensure that the traffic-skew is maintained after deleting l_1 and all flows crossing it.

For any pair of links $l_2, l_3 \neq l_1$, we have $T(\{l_2, l_3\})/T(\{l_2\}) \leq \frac{1}{\alpha}$. Let's say that the number of packets crossing link l_2 is $T'(\{l_2\})$ after we delete l_1 and all flows crossing l_1. Then we have,

$$
\begin{aligned}
\frac{T'(\{l_2, l_3\})}{T'(\{l_2\})} \leq &\ \frac{T(\{l_2, l_3\})}{T'(\{l_2\})} && (3.40)\\
= &\ \frac{T(\{l_2, l_3\})}{T(\{l_2\}) - T(\{l_1, l_2\})} && (3.41)\\
\leq &\ \frac{T(\{l_2, l_3\})}{(1 - 1/\alpha)T(\{l_2\})} && (3.42)\\
= &\frac{1}{\alpha - 1} && (3.43)
\end{aligned}
$$

Thus, for subsequent iterations, after deleting l_1, traffic is $1/(\alpha - 1)$-skewed and the number of failures is $\alpha/2 - 1$ in the deleted graph. The same argument as before shows that the greedy algorithm will pick a failed link in every iteration as long as there is at least one failed link not in the current hypothesis.

Stopping Criteria: Finally we need to show that once all failed links are picked, the greedy algorithm will halt. This happens when $LL(H_l) < 0$ for all l not in the current hypothesis. Note that the input to log likelihood computations in the current iteration is the topology obtained after deleting all links in the current hypothesis and all flows crossing any of those links. As before we have,

$$E[LL(H_l)] = \sum_{f \in F(l)} E[X(f, H)] = \sum_{f \in F(l)} n(p_f - \mu) \tag{3.44}$$

Note that for a path with all good links, $p_f - \mu \leq p_g - \mu < 0$. Hence, $E[LL(H_l)]$ is bounded away from 0 towards $-\infty$. Since, $LL(H_l)$ is the sum of independent binary variables corresponding to packets each of whose expectation is $\leq (p_g - \mu) < 0$, applying Chernoff bounds followed by a union bound for all links shows that $LL(H_l) < 0$ for all links l with high probability. This completes the proof of Theorem 3.2. QED.

3.11 SUMMARY

Flock is a fault localization system for large datacenter networks based on end-to-end information. Flock's key innovation is an optimized MLE inference algorithm which allows it to use a PGM at scale, achieving both high accuracy and speed, where past work achieved only one of the two.

CHAPTER 4: DIAGNOSING GREY FAILURES VIA MICRO-ACTIONS

4.1 INTRODUCTION

Datacenter networks experience significant downtime due to silent gray failures [59]. For instance, a flow could be getting dropped due to a wrong or a corrupted routing table entry at a faulty switch [50] (commonly known as a 'network black hole'). Or a switch might be silently dropping a small fraction of the packets [90] resulting in poor observed performance for some flows. Locating the fault given observed performance symptoms is hard in a Clos network with several symmetric paths and limited visibility inside the datacenter.

Effective diagnostic tools [103, 60, 88] for grey failures have been designed for programmable switches which provide increase visibility into the network. However, programmable switches have not been generally deployed ($<$ 13% estimated adoption in 2023 [52], see also [5]) and most enterprise/cloud networks primarily comprise of traditional switches. Even if they are used in some datacenters, they might be deployed limitedly. For these reasons, designing effective diagnostic tools for non-programmable switches is an important research objective.

In practice, then, troubleshooting grey failures happens roughly as follows, in two stages. First, operators can collect passive monitoring data, which is the most practical and commonly available telemetry with network monitoring tools. Normally, this passive telemetry contains only end-to-end statistics about flows (e.g. retransmissions, latency etc.) and not the path taken by flows. Clos networks with several ECMP multipaths pose a problem in disambiguating symmetric links with passive monitoring, since the path is obscured (see figure 4.1). Previous automated fault localization based on passive monitoring [50, 54, 13] can only localize the problem to a relatively large set of components such as a tier or a pod of switches and indeed, symmetries in the network mean it is not possible to always localize faults completely. Thus, regardless of whether the operator is manually inspecting passive flow telemetry or using a more automated method, this stage will often still result in a large set of potential culprits.

Second, since the fault typically can't be localized with passive telemetry alone, operators must troubleshoot manually, often via ad-hoc modifications to the network. For example, working with the set of potential culprits that remains after inspecting passive monitoring, the operator may disable a suspect link, reboot a router, or change routing rules for select flows, in the hope of finding the fault. Locating the fault via this manual exploration, one component at a time, can be time consuming (even occupying a period of weeks, according

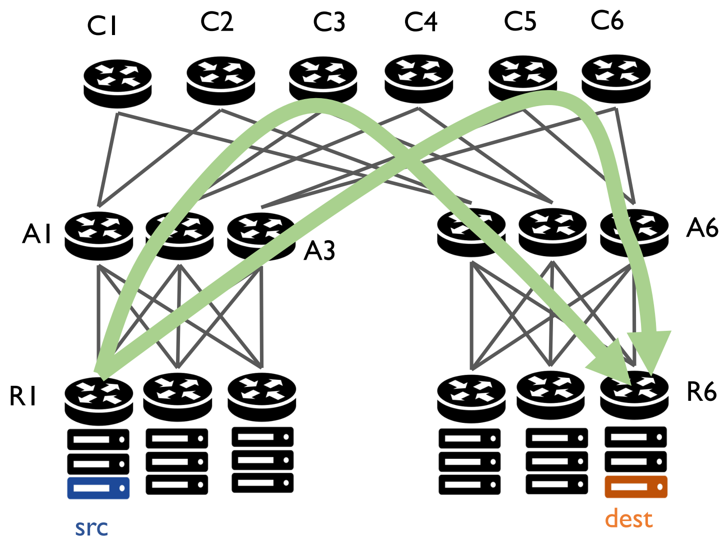


Figure 4.1: Symmetric links in a Clos topology make it very hard to localize faults completely. Consider a failure scenario where several flows from src → dest are getting dropped due to a network "black hole" at one of the core switches (C1 – C6). It's impossible to distinguish between C1-C6 without knowing the exact ECMP path taken by the flows. Two such ECMP paths are shown: any flow that can possibly take one path can also take the other.

to our discussions with operators). Furthermore, invasive changes can introduce risk, e.g., leading to insufficient capacity for some traffic [105, 75] or routing policy violations.

Our goal is to design practical tools to enable accurate localization for grey failures in datacenter deployments. For increased applicability, we restrict ourselves to telemetry from passive monitoring for non-programmable switches since that is the most common environment. To accomplish our goal, we propose a new method of diagnosing network grey failures– by iteratively making a small, but carefully planned sequence of *micro-actions* to the network that otherwise operators take in an ad-hoc way and then running automated inference to analyze the effect of those changes to localize the fault. These micro-actions are lightweight, and might include removing a path from the set of ECMP paths for some select flows or modifying ECMP weights or adding a custom route for some specific (*source*, *destination*) host pair.

There are two challenges in designing a diagnostic tool based on the above approach. One, how do we algorithmically decide what sequence of micro-actions would aid localization while minimizing the number of steps, without actually rolling out the changes? And two, what inference algorithm should we use to localize the fault using the combined telemetry from all the steps?

We design and prototype a diagnostic tool for network black-holes called FaultFerence, that generates a short sequence of micro-actions, from a menu of allowed changes, using an algorithm that minimizes the number of changes via topological analysis of the network (§ 4.2.1.2). FaultFerence then combines passive monitoring telemetry from all the micro-action steps and runs a Bayesian inference algorithm [54] to produce the faulty component(s). In our experiments, FaultFerence was effective in localizing the problem to 2 or fewer components[18] in $> 98\%$ of cases, while reducing the number of micro-action steps by $2.1\times$ from 19 to ≈ 9 on average, compared to a baseline semi-automated scheme.

Overall, our results show that utilizing micro-actions is a promising and practical approach to localizing grey network failures using passive monitoring. Our work opens opportunities to design other practical tools to detect various problems such as high latency [83] and silent packet drops [90] in real deployments. It also opens the possibility of designing systems that take advantage of even less intrusive micro-actions (e.g. modification in ECMP weights, packet mirroring for a small subset of packets), whose effects would ordinarily be too subtle for an operator to manually interpret but which an automated inference algorithm can make sense of. Via use of passive monitoring, we believe this line of work can have real impact for network management in datacenter deployments. We plan to make our code available open source post publication.

4.1.1 Motivation

A *grey failure* occurs when a faulty component does not directly report that it has failed [59]. Examples of grey failures include silent packet drops because of corruption in TCAM forwarding table entries [66, 104], a misconfigured switch causing high latency [104, 83], inter-switch or inter-card drops [103]. Grey failures are very challenging to detect, requiring multiple hours or even days [104] for diagnosis. For e.g., silent packet drops constituted 50% of faults that took > 3 hours to diagnose in [103].

Past works have designed tools to help troubleshoot network grey failures. Tools such as Omnimon [60], FANcY [73] and NetSeer [103] use programmable switches to gain more visibility into the network. For instance, to track dropped packets, NetSeer runs a packet sequencing protocol in the programmable switches' hardware, across all links. These tools are effective in diagnosing grey failures. However, programmable switches have limited adoption (<13% estimated adoption in 2023 [52]) and are not universally deployed across the fleet in most datacenters. Tools such as Everflow [104] use packet mirroring to track the path of

[18]With the micro-actions we considered, fundamentally it wasn't possible to narrow it down to less than 2 components in some cases.

some packets. Everflow can work with non-programmable switches, but has low coverage of failure events with sampling (<1% coverage [103]). Everflow is not scalable if mirroring is enabled for all packets.

Another set of tools run active probes and collect measurements about those probes, optionally employing source routing for the probe packets [90, 24, 57]. Sending active probes may have extra overheads, at the end-hosts and the network. (In our discussions with operators, the greatest concern is running probes on end-hosts since even very small amounts of interference with customer workloads is highly undesirable.) More importantly, active probes may fail to reproduce failures since they take a different data path in the switch routing table than the application traffic [90]. For instance, while packets belonging to regular traffic might be getting dropped at a device due to a corrupted TCAM table entry [104], the probe packets invoking some other uncorrupted table entry may not get dropped.

The most practical and commonly available form of telemetry comes from passive monitoring of application flows. Designing automated fault diagnosis tools for this telemetry is therefore an important research goal. With passive monitoring, the system tracks flow-level performance and reports flow metrics such as retransmissions, flows dropped, packets sent, etc. Usually, the topology is also known and one can obtain the paths that a flow from a *src* host to a *dst* host can take. Note that datacenters employ ECMP to balance traffic for any (*src*, *dst*) pair among several shortest paths. The exact path taken by flows is however hard to obtain with passive telemetry. As a result, multipath ECMP routing poses a problem for fault diagnosis with passive monitoring. We illustrate this with an example.

Consider the failure scenario in Figure 4.1 where a significant fraction of flows for a particular (src, dest) pair are getting dropped because of a network "black hole" [50] at one of the core switches (C1 – C6). Because of symmetry, it is impossible to distinguish which core switch (C1-C6) might be dropping packets without knowing the exact ECMP path taken by the flows. Obtaining the path taken by each flow is hard in real deployments as the path is decided in real time via hashing on the packet's 5-tuple (src ip, dst ip, src port, dst port, protocol) with a time-seeded hash function chosen by the device vendor and not generally known. Such symmetries can exist even in irregular topologies and multiple devices in the path set for (src, dest) might fall in the same equivalence class.

Thus, for passive telemetry with path uncertainty [50, 54] (exact path taken by a flow is unknown, the set of possible paths is known), existing fault localization tools can only partially localize the problem to a relatively large set of components (e.g. tier, switch pod) because of classes of indistinguishable links. In the absence of good automated diagnosis tools, network operators then have to resort to haphazardly making temporary micro-actions in the hope of learning something from the monitoring telemetry to diagnose the fault. These

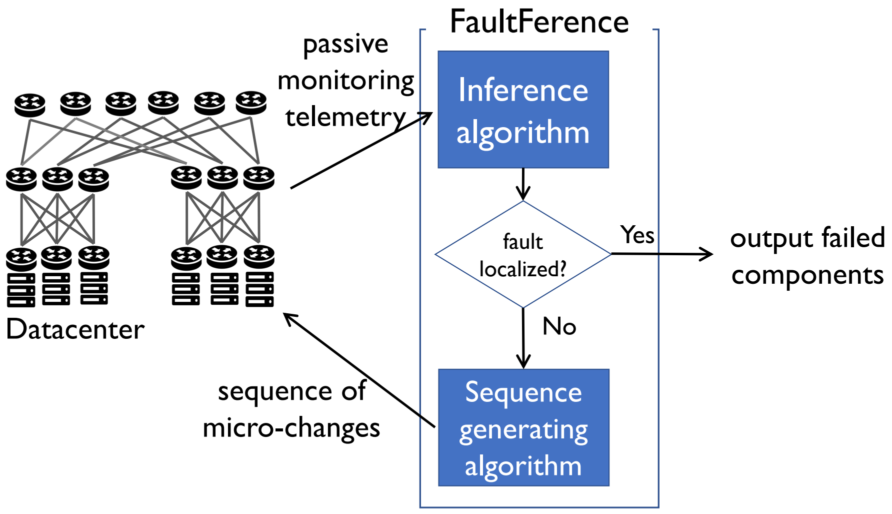


Figure 4.2: System diagram of FaultFerence

micro-actions might be rolled out for select traffic and can be of various types- taking down a link, modifying ECMP weights, installing custom routes, packet mirroring, etc.

Rolling out such micro-actions in an ad-hoc way can have undesired consequences. First, localizing failures by manually making sense of the micro-actions can be time consuming, requiring several tries which can additionally increase invasiveness of the diagnosis procedure. Second, relying on a human to decide on what micro-actions to make can be unsafe. For instance, turning off a link, even for select traffic, can reduce the capacity between two hosts beyond what's acceptable when some links in the network are already turned off because of previously identified faults [105]. It is easy for an algorithm to check for policy and capacity violations before making a micro-action.

Thus, it is desirable to have a diagnostic approach for localizing network faults based on passive monitoring that systematically makes micro-actions that otherwise the operators would have taken in an ad-hoc way. Such an approach can optimize the number of micro-action steps required for accurate localization, thereby reducing diagnosis time and making the diagnosis less invasive than a manual procedure. Finally, using automated probabilistic inference also allows us to make use of less invasive micro-actions (e.g. changing ECMP weights for some flows) that humans would generally find difficult to reason about in a manual workflow.

4.2 DESIGN

The first step of FaultFerence's diagnosis is running inference over monitoring data from the unchanged network. In this step, FaultFerence localizes the fault to an equivalent set ($\mathcal{S}$) of devices. It then (1) estimates what sequence of micro-actions to roll out to ensure good localization within $\mathcal{S}$, in a few steps (without actually rolling out the changes). Once the micro-actions have been rolled out (and unrolled back) (2) FaultFerence's inference algorithm ingests the passive monitoring telemetry from all steps and runs inference on the combined telemetry and localizes the fault, running more iterations of micro-action steps if required (looping back to (1)). Figure 4.2 illustrates the FaultFerence system. We describe the design of these parts next.

4.2.1 Generating sequence of micro-actions

Information about packets transiting a slightly different set of paths can reveal information about the location of the network fault. But what kind of micro-actions will *(a) aid fault localization* and *(b) minimally affect production traffic?* We first discuss what type of micro-actions can be useful and acceptable.

4.2.1.1 Types of micro-actions

Clearly, (b) requires careful design so as to not bring down network capacity or harm otherwise performant flows or cause routing violations. For instance, one strategy could be to temporarily take down a single link to block certain routes to check if that fixes this problem. However, taking down an entire link can be an expensive operation (in terms of time and traffic impact) since it requires draining the links of all their current traffic and then the distributed routing protocol to re-converge to a new routing state.

Our current prototype incorporates two types of micro-actions- (1) Temporarily taking down paths, that include a particular link, for the problematic (src, dest) pair (as in figure 4.1). and (2) modifying ECMP weights for paths for the problematic (src, dest) pair. Both of these micro-actions are less expensive than completely taking down a link since (a) only a small subset of traffic is affected and (b) the routing protocol does not need to re-converge.

For the type of micro-actions discussed above, how do we generate the sequence of micro-actions to be rolled out to the network? We discuss an example before describing the concrete algorithm.

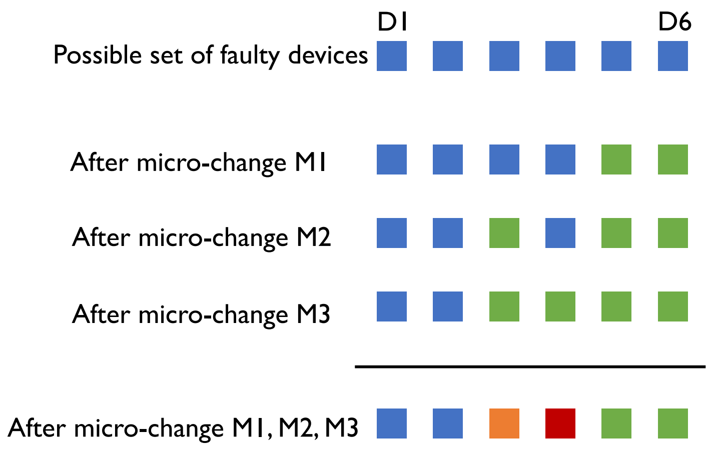
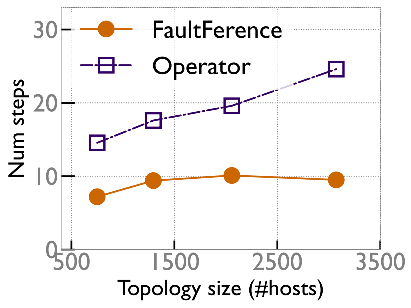


Figure 4.3: FaultFerence's micro-action sequence generating algorithm: without any micro-actions, there are 6 devices (D1 - D6) any of which might be faulty. After micro-action M1, we estimate that the faulty device can be localized to one of the two sets (shown in blue and green). Similarly, for micro-actions M2 and M3, we estimate the "equivalent" sets after they will be rolled out (without actually doing so). We estimate that the faulty device can be localized to one of four sets (shown in blue, orange, red and green) after M1, M2, M3. Using this procedure, we measure how good is the sequence M1, M2, M3 by computing the expected size of the "equivalence class" of faulty links at the end.

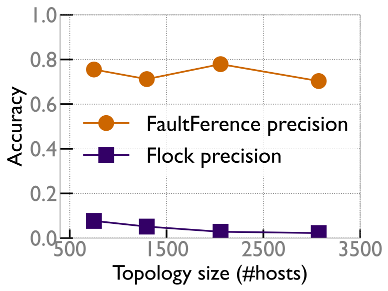
In our running example from Figure 4.1, one option is to add a temporary high priority rule at switch R1, so that packets for the problematic (src, dest) pair do not get forwarded to A3 from R1. If the problem persists after this change, then switches A3, C5, C6 and A6 can be ruled out. If not, then one of A3, C5, C6 and A6 is likely to be the faulty switch. We could repeat the same for switches A1 and A2, by installing rules at R1 so that packets for (src $\rightarrow$ dest) do not get forwarded to A1 and A2. This gives us a series of micro-actions to carry out, first for A1, then for A2, then for A3.

Using the above procedure, say we narrow down the faulty device to {A3, C5, C6, A6}. To localize the problem further, we can employ a series of micro-actions at the aggregation layer (A3 and A6) and then at the core layer (C5, C6). Adding a rule at A3 to avoid C5 can rule out C5. C6 can be ruled out similarly.

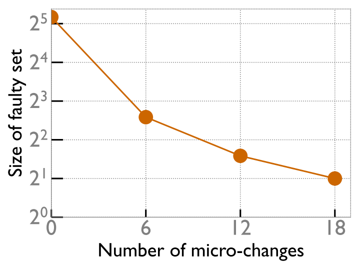
In general, how do we generate a sequence of micro-actions for any topology and problem? If we assume a perfect Clos topology with perfect ECMP-based routing, then the series of steps is similar to the above example, first we eliminate switches by installing a rule in the ToR switch, then installing similar rules in the aggregation layer switches. But real world topologies have irregularities. Hence, we design an algorithm to generate the micro-action steps automatically given an arbitrary topology. We describe this algorithm next.



(a) FaultFerence: number of micro-action steps compared to a semi-automated baseline for Clos topologies with different sizes. Note that the number of steps increases slowly for FaultFerence.



(b) Comparing state-of-the-art passive fault localization (Flock) with no micro-actions to FaultFerence. They both achieved high recall (>98%), hence we only show precision.



(c) How FaultFerence reduces the size of the equivalence class of faulty devices with increasing number of micro-actions. We used micro-actions that modify ECMP weights for this experiment.

Figure 4.4: Experiments showing (a) how FaultFerence localizes the fault in a small number of micro-action steps, (b) how FaultFerence enables diagnosis of failure scenarios that traditional fault localization is unable to diagnose via passive telemetry and (c) how the set of potential culprits reduces with the number of steps with FaultFerence.

4.2.1.2 Algorithm to generate micro-action sequence

Assume that before any micro-action, we know that one device from the set $S = \{D_1,\ D_2,\ \ldots\ D_n\}$ is faulty, via the initial analysis. We want to decide a sequence of micro-actions to roll out to the network that will narrow down the failure further. We assume that the network topology is known but it may be arbitrary. To do so, first, we describe a way to evaluate how good a given sequence of micro-actions, say M_1, M_2, M_3 is (without actually rolling them to the network). Here, an example of a possible micro-action could be adding a rule at $R1$ to avoid $A3$ for packets corresponding to a particular (src, dest) pair.

We try to estimate how would the set S of faulty devices get narrowed down after each of the micro-actions M_1, M_2 and M_3. To do so, we estimate which paths the flows would take after a micro-action (say M_1), and which sets of devices can be differentiated based on that. This would effectively give us "equivalent sets" of faulty devices such that if we were to roll out M_1, we will be able to localize the fault to one of the equivalent sets (refer to Figure 4.3), depending on where the fault actually lies.

Once we have the estimated equivalent sets corresponding to each micro-action $(M1, M2, M3)$, we take the intersection of all the equivalent sets to get final equivalent sets (as shown in Figure 4.3). This procedure tells us that after carrying out $M1, M2, M3$ separately one by one, we should be able to localize the problem to one of the equivalent sets using monitoring information obtained from each of those steps.

Finally, to quantify how good a sequence of micro-actions M_1, M_2, M_3 is, we assign an information theoretic score to the final equivalent sets. Let's say there were n devices, any one of which could be faulty. If those n devices are partitioned into equivalent sets $S_1, S_2 ... S_k$ after M_1, M_2, M_3, then the score is given as:

$$Score([M_1, M_2, M_3]) = \sum_{i=1}^{k} -\frac{|S_i|}{n} \log \frac{|S_i|}{n} \tag{4.1}$$

We define the score in this way since it represents the information contained in the equivalent sets $S_1, S_2 ... S_k$ [3].

We generate the sequence of micro-actions by greedily picking the best micro-action at any given step, that maximizes the information-theoretic score of the equivalent sets obtained after rolling all the micro-actions till now.

Finally, FaultFerence terminates when it has localized the fault to a set of components and it is unable to localize it any further (when the equivalence class of faulty devices remains unchanged in 3 consecutive steps).

4.2.2 Analysis Algorithm

How do we analyze the monitoring data obtained after each modification? One option is to examine the telemetry manually for instance, by looking at the dropped flows. A more principled approach would be to reason about this algorithmically. We can use a suitable fault localization algorithm that handles telemetry for passive monitoring, such as Flock [54] or Score [67]. We use Flock with passive monitoring in our prototype as its Bayesian inference via PGMs achieves high accuracy. We calibrated Flock's hyperparameters as done in [54].

4.3 EVALUATION

4.3.1 Setup

Network simulator: We use the Python datacenter simulator from [54] which outputs a trace consisting of flow metrics (packets sent, retransmissions, flow-dropped?) for an input network topology and traffic. We use standard uniform traffic with Pareto-distributed flow sizes (mean 200KB) [21] to mimic irregular flow sizes in the datacenter.

FaultFerence prototype: Our current implementation of FaultFerence talks to the simulated datacenter (see figure 4.2), generating micro-actions to the topology and collecting measurements from the simulator with the new topology after a micro-action is rolled out. We implemented FaultFerence's sequence generating algorithm described in § 4.2.1.2. FaultFerence then runs Flock's inference algorithm on the aggregated measurements from all micro-action steps to find an equivalent set of faulty devices. Depending on whether the failure has been localized or not, it can rerun the sequence generating algorithm to get another sequence of micro-actions.

We consider two types of micro-actions - (a) disabling a link for some specific (src, dst) pairs and (b) changing ECMP weights for some specific (src, dst) pairs. For each micro-action step, FaultFerence runs the simulator with the updated topology to generate performance reports of 2000 flows.

Topologies: We use the standard 3-layer Clos topologies (FatTrees) [20] with different sizes (with K=10, 12, 14 and 16). For all topologies, we use an oversubscription of 3 at the ToRs.

Failure scenarios: We consider the problem of detecting network black holes at a switch [50] where the faulty device drops all packets from a particular source to a destination host. We randomly choose the faulty device, the source host and destination host for the fault. For diagnosing black holes, Flock requires information about which flows got dropped and which ones where successful.

Baseline: To the best of our knowledge, we are not aware of any scheme that iteratively narrows down a fault using micro-actions. Hence, for the baseline we considered what an intelligent operator might do. The operator begins with all set of links in the ECMP path set for the problematic (src, dst) host pair. Then, they randomly remove a link l from that path set to check whether that makes the problem go away or not. If the problem does not go away, they can eliminate l and any link that participates in paths that all have l (call this set L_e). If the problem goes away after removing l, they can narrow down the problem to L_e. This constitutes one step of the diagnostic procedure. Then, on the reduced set of links, the operator repeats the procedure till the problem can not be localized any further (when the equivalence class of faulty devices does not change for 3 consecutive steps).

Metrics: We quantify the number of micro-action steps required by FaultFerence for diagnosis. The number of steps directly translate to the diagnosis time and the impact caused by the micro-actions in the datacenter. We also use *precision* and *recall* to quantify accuracy of the diagnosis procedure. Precision is the fraction of predicted failed devices that actually failed and recall is the fraction of failed devices that were correctly predicted as failed.

4.3.2 FaultFerence vs Operator

4.3.2.1 Number of steps

As discussed in § 4.2.1.2, FaultFerence improves the time taken to diagnose the faulty link by rolling out a carefully planned sequence of micro-actions. Figure 4.4a shows the number of steps taken by the Operator baseline and FaultFerence over different topology sizes to diagnose the fault. For this experiment, we used micro-actions that remove a link from the ECMP path set of a particular (src, dest) host pair. As figure 4.4a shows, the number of steps for FaultFerence grow slowly while the number of steps for the baseline grows roughly linearly with the topology scale. This highlights the benefit of using FaultFerence's algorithm to plan a small sequence of micro-actions. Note that a smaller number of steps translates to reduced diagnosis time and reduced impact on the network from the changes.

4.3.2.2 Recall

FaultFerence had a recall of 98.7% among the 80 runs across all the topologies. On the other hand, the operator baseline we implemented is guaranteed to have 100% recall. Note that any probabilistic inference algorithm will make some mistakes and some inaccuracy is expected because of that.

4.3.2.3 Precision

Figure 4.4b shows FaultFerence's precision for localizing the fault. For comparison, we have also shown Flock, which does not make any micro-actions and hence relies on telemetry from the unchanged network. Flock had a precision of 0.022 for the largest topology in our experiments. This leaves a lot of work to the operator, requiring them to localize the fault from $>$ 40 devices. In comparison, FaultFerence narrowed down the fault to 1 or 2 devices after a few micro-actions.

4.3.3 Changing ECMP weights for diagnosis

Similar to disabling the link for a particular (src, dest) host pair, another type of micro-action could be to modify the ECMP weights at a device for the (src, dest) pair. Such changes are typically hard for the operator to reason about or use while debugging since it requires some form of statistical inference. Using probabilistic reasoning of Flock allows FaultFerence to use less invasive micro-actions like modifying ECMP weights. Figure 4.4c shows how FaultFerence reduces the size of the equivalence class of faulty devices with the number of micro-action steps. Overall, with this type of micro-action, FaultFerence had a precision of 0.41 and recall 1 at the end of the diagnosis procedure.

4.3.4 Running time

There are two sources of runtime complexity for FaultFerence. First, the data collection time when the micro-actions are temporarily rolled out and monitoring telemetry is gathered. This depends on the traffic load and the reporting interval of the monitoring system. We estimate that the data collection time should be in the order of a few minutes per micro-action step. The second source of runtime are the FaultFerence's sequence generating algorithm and the inference algorithm. Combined, they took < 30 seconds with 8 cores in our current implementation, but there's room to optimize it further.

4.4 RELATED WORK

Systems that make network changes: Swarm [75] explores mitigations for an already identified fault and can be complementarily used with FaultFerence. It automatically checks for capacity and policy violations, which can also be used for FaultFerence. We leave this for future work. NetPilot [93] takes a trial and error approach to failure localization, trying out operator actions to mitigate the fault. Unlike FaultFerence, it does not use automated inference for localizing the failure and thus might require a significantly longer duration to diagnose the fault. Corropt [105] takes as input, links with high packet corruption rates, and takes actions to mitigate the failures, while ensuring a threshold capacity. Unlike Fault-Ference, it does not run inference to localize the failure or generate a sequence of changes to roll out to the network.

Other failure inference systems: Prefix [99] learns patterns from syslogs to predict future failures. This is a different approach to mitigating the effect of failures, but does not help in diagnosing known issues. deTector [79] uses a greedy heuristic for computing a small number of links that explain the probe losses analogous to MaxCoverage [66]. It plans the set of active probes to send, but does not plan for changes to make to the network. SwitchPointer [89] proposes an in-network scheme for fault diagnosis. However, it is unsuitable for today's networks as it requires intrusive changes to the network hardware. LightGuardian collects per-hop flow-level information for all flows using a lightweight mechanism that embeds a small data structure into the packets. However, LightGuardian relies on programmable switches and thus falls outside our goals.

4.5 LIMITATIONS AND FUTURE WORK

Our work opens new research directions to explore-

New types of micro-actions: Our current prototype explores two micro-actions – deleting routes or modifying ECMP weights for certain source-destination pairs. Exploring other changes, such as adding new routes or selective packet mirroring [104] are promising directions for future work. Another interesting direction could be to design even less invasive micro-actions, that possibly are too subtle for an operator to make sense of, but can be reasoned about using an algorithm (e.g. very minimal change in ECMP weights).

Failure scenarios: Our prototype dealt with network black holes. Similar techniques can also be applied to diagnose other failure scenarios such as high latency [83], silent packet drops [90] or load imbalance [104].

Beyond Clos topologies: We evaluate FaultFerence on Clos topologies which have a high degree of symmetry. It can be even more beneficial for irregular topologies, where it would be harder for the operator to manually reason about the topology to come up with a sequence of micro-actions.

Beyond datacenter networks: We focused on datacenter networks; it would be important future work to explore diagnosis tools for WAN and IP networks based on the idea of micro-actions.

Failure diagnosis in other domains: Automated diagnosis among several components is a challenge in several distributed systems e.g. diagnosing high service times in a distributed application [30]. Employing the approach of making micro-actions to the system to aid failure inference can be a promising direction to design effective failure diagnosis tools for such systems.

CHAPTER 5: CONCLUSION AND FUTURE WORK

In this thesis, we described our work in designing systems, models and algorithms for root cause analysis that can handle the complexities of modern distributed and networked infrastructure such as scale, complex inter-dependencies between entities and the coarse grained nature of commonly available telemetry in real environments. To do so, we designed probabilistic graphical models (PGMs) designed specifically for the troubleshooting task at hand, along with custom-designed inference algorithms to handle scale and other challenges of that environment.

In Murphy, we designed a Markov Random Field (MRF) based model to predict causality between entities. Murphy's MRF model precisely captures the weak non-causal dependency information between entities, which leads to a more accurate analysis. The input monitoring data that Murphy utilizes is universally present in most networked systems, making Murphy applicable in a lot of environments. To handle the large scale of our environments comprising many entities, we introduced a counter-factual reasoning algorithm for MRFs, that looks at a small set of entities that are likely going to be relevant for root cause analysis. Using simulations and a real world dataset of incidents, we showed that Murphy outperformed previous approaches for reasoning about performance-related incidents.

In Flock, we designed a Bayesian network to infer hidden information about devices in a datacenter network such as the number of unaccounted packet drops. The Bayesian network model was able to capture the complexities of the environment such as the lack of knowledge of paths taken by flows in a datacenter. A well-known problem associated with inference via Bayesian networks is their intractability for large problem sizes, such as in a datacenter. To alleviate this problem, we designed algorithmic techniques to speed up inference in discrete-valued PGMs that led to up to four orders of magnitude faster inference over previous PGM-based solutions. Using extensive experiments in simulation and testbed, we demonstrated the benefits in inference accuracy due to PGM-based modeling, over previous datacenter fault localization schemes. Our algorithmic techniques to scale up inference were critical in unlocking these benefits for large datacenter networks.

In FaultFerence, we designed a closed loop fault localization system for datacenter networks consisting of symmetric components which all perform the same function. In environments where the path taken by flows is unknown, it is impossible to distinguish between sets of symmetric devices using monitoring data from end-hosts. FaultFerence runs iterations of inference followed by microactions that very slightly tweak the state of the network. FaultFerence's inference is based on a Bayesian network model (such as the one used in

Flock), which helped us to capture the path uncertainty of flows in the datacenter. To pick microactions that would break symmetry of the localization problem, FaultFerence used a custom-designed algorithm. Using simulation experiments, we showed that FaultFerence effectively localizes failures and reduces the time and invasiveness of the ad-hoc localization procedures employed today.

Across these systems, we discovered that high fidelity modeling of the system results in better diagnosis. In Murphy, staying true to weak inter-dependencies between entities present in the monitoring data (rather than trying to enforce strong causal dependencies) resulted in a better model and better diagnosis. In a datacenter network, accurately modeling the path uncertainty of flows and the noisy occasional packet drops resulted in higher inference accuracy. We believe that some of our techniques are more general than the specific realizations in this thesis. Hence, they can be applied to other problem scenarios–

- Applying Bayesian network based modeling to handle failures of virtual components that seem local, but are actually over the network, such as virtual disk failures [98]. Since the number of such components and the possible number of paths that connect them to their users can be large, our techniques to speed up inference can be used to make PGM-based root cause analysis feasible in such cases.

- Applying Markov Random Field based modeling to capture weak dependencies in other contexts such as in a software system with several components (software blocks, resources, threads etc.) and using such a model to capture faults in that system.

- Using PGM-based models to reason about hypothetical performance scenarios. For instance, how would the latency of a service get affected if an additional replica is added? To reason about such a scenario, a system needs to model how the service latency depends on other components such as the downstream services in the distributed system or other hardware components (e.g. the network).

- Using closed loop inference, similar to FaultFerence's design, to capture bugs in distributed and software systems. For instance, for detecting erroneous service components in a distributed system comprised of several services, one strategy could be to very slightly change the request routing, so that some requests go to a different replica and then infer the culprit service using a suitable inference algorithm.

Finally, we believe that our work only scratches the surface in automated troubleshooting of distributed and networked systems. Ensuring reliability of such systems is a major goal for enterprises. Hence, establishing more fundamental ways to model the system and training that model using available data for high quality inference remains a fertile ground for research. Using the lessons learnt in our thesis, we believe that it's important to stay true to the information present in the input monitoring telemetry in this pursuit.

REFERENCES

[1] Cadvisor. https://github.com/google/cadvisor.

[2] Calculating cost of downtime. https://www.atlassian.com/incident-management/kpis/cost-of-downtime.

[3] Entropy (information theory). https://en.wikipedia.org/wiki/Entropy_(information_theory).

[4] Hourly cost of downtime. https://itic-corp.com/tag/hourly-cost-of-downtime/.

[5] Intel exits another non-core business. https://www.fool.com/investing/2023/01/29/intel-exits-another-non-core-business.

[6] Jaeger. https://www.jaegertracing.io/.

[7] Murphy: Deathstarbench traces. https://github.com/netarch/Murphy-traces.

[8] One in ten rule of thumb. https://en.wikipedia.org/wiki/One_in_ten_rule.

[9] VMware Aria Operations for Networks. https://www.vmware.com/in/products/aria-operations-for-networks.html.

[10] VMware Aria Operations for Networks - Working with Application Discovery. https://docs.vmware.com/en/VMware-Aria-Operations-for-Networks/SaaS/Using-Operations-for-Networks/GUID-71D10285-1CA6-4F85-A5D8-01B0BEEF3BC2.html.

[11] wrk2. https://github.com/giltene/wrk2.

[12] Zipkin. https://zipkin.io/.

[13] Netnorad: Troubleshooting networks via end-to-end probing. https://code.fb.com/networking-traffic/netnorad-troubleshooting-networks\-via-end-to-end-probing/, 2016.

[14] In-band network telemetry (int) dataplane specification. https://github.com/p4lang/p4-applications/blob/master/docs/INT_latest.pdf, 2020.

[15] Flock code. https://github.com/netarch/FaultLocalization, Github.

[16] Pf_ring by ntop software. github.com/ntop/PF_RING, Github.

[17] Manage engine traffic analyzer.
`https : / / www . manageengine . com / products / netflow/`, URL.

[18] Solarwinds traffic analyzer.
`https : / / www . solarwinds . com / netflow-traffic-analyzer`, URL.

[19] Abhishta Abhishta, Roland van Rijswijk-Deij, and Lambert J. M. Nieuwenhuis. Measuring the impact of a successful ddos attack on the customer behaviour of managed dns service providers. *SIGCOMM Comput. Commun. Rev.*, 48(5):70–76, jan 2019.

[20] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[21] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 253–266, San Jose, CA, 2012. USENIX.

[22] Andrew Lerner. Inclusion Criteria for the 2016 NPMD Magic Quadrant.
`https : / / blogs . gartner . com / andrew-lerner / 2015 / 06 / 29 / gotnpmd/`, 2015.

[23] Arista. Arista Network Telemetry. `https : / / www . arista . com / en / solutions / software-defined-network-telemetry`, Accessed 2021-01-27.

[24] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 419–435, Renton, WA, April 2018. USENIX Association.

[25] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 13–24, New York, NY, USA, 2007. ACM.

[26] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. PINT: probabilistic in-band network telemetry. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 662–680, 2020.

[27] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[28] British Telecommunications. Contract for BT Managed WAN Services. `https://business.bt.com/content/dam/terms/it-solutions-support/bt1190.pdf`, 2018.

[29] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol (SNMP). In *RFC 1157*. Internet Engineering Task Force, 1990.

[30] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 115–128, New York, NY, USA, 2016. Association for Computing Machinery.

[31] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 115–128, New York, NY, USA, 2016. Association for Computing Machinery.

[32] Xu Chen, Ming Zhang, Z. Morley Mao, and Victor Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *OSDI*, January 2008.

[33] Yan Chen, David Bindel, Hanhee Song, and Randy H. Katz. An algebraic approach to practical and scalable overlay network monitoring. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 55–66, New York, NY, USA, 2004. ACM.

[34] Cisco. Monitoring and Troubleshooting With Cisco Prime LAN Management Solution 4.1. `https://www.cisco.com/c/en/us/td/docs/net_mgmt/ciscoworks_lan_management_solution/4-1/user/guide/monitoring_troubleshooting/mnt_ug/SNMPInfo.html`, 2018.

[35] Cisco. Cisco Bug: CSCvn56156 - Silent packet drops may occur on FXOS platforms due to classifier table entry corruption. `https://quickview.cloudapps.cisco.com/quickview/bug/CSCvn56156`, 2020.

[36] Cisco. Configure Link Flap Prevention on a Cisco Business Switch using CLI. `https://www.cisco.com/c/en/us/support/docs/smb/switches/Cisco-Business-Switching/kmgmt-2249-configure-the-link-flap-prevention-settings-on-a-switch-thro.html`, July 2020.

[37] Benoit Claise. Cisco Systems NetFlow services export version 9. *RFC 3954 (Internet Standard), Internet Engineering Task Force*, 2004.

[38] Benoit Claise, Brian Trammell, and Paul Aitken. Specification of the ip flow information export (ipfix) protocol for the exchange of flow information. *RFC 7011 (Internet Standard), Internet Engineering Task Force*, 2013.

[39] Amogh Dhamdhere, Renata Teixeira, Constantine Dovrolis, and Christophe Diot. Netdiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. In *Proceedings of the 2007 ACM CoNEXT Conference*, CoNEXT '07, New York, NY, USA, 2007. Association for Computing Machinery.

[40] Divya Rao. Hot off the press: Introducing OpenConfig Telemetry on NX-OS with gNMI and Telegraf! `https://www.cisco.com/c/en/us/td/docs/net_mgmt/ciscoworks_lan_management_solution/4-1/user/guide/monitoring_troubleshooting/mnt_ug/SNMPInfo.html`, July 2020.

[41] Pradeep Dogga, Karthik Narasimhan, Anirudh Sivaraman, Shiv Saini, George Varghese, and Ravi Netravali. Revelio: Ml-generated debugging queries for finding root causes in distributed systems. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 601–622, 2022.

[42] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1(4):397–413, August 1993.

[43] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *12th {USENIX} symposium on networked systems design and implementation ({NSDI} 15)*, pages 469–483, 2015.

[44] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: Practical and scalable ml-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 135–151, New York, NY, USA, April 2021. Association for Computing Machinery.

[45] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud amp; edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[46] Jiaqi Gao, Nofel Yaseen, Robert MacDavid, Felipe Vieira Frujeri, Vincent Liu, Ricardo Bianchini, Ramaswamy Aditya, Xiaohang Wang, Henry Lee, David Maltz, Minlan Yu, and Behnaz Arzani. Scouts: Improving the diagnosis process through domain-customized incident routing. In *Proceedings of the Annual Conference of the*

ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '20, page 253–269, New York, NY, USA, 2020. Association for Computing Machinery.

[47] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. SIMON: A Simple and Scalable Method for Sensing, Inference and Measurement in Data Center Networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 549–564, 2019.

[48] Denisa Ghita, Hung Nguyen, Maciej Kurant, Katerina Argyraki, and Patrick Thiran. Netscope: Practical network loss tomography. In *Proceedings of the 29th Conference on Information Communications*, INFOCOM'10, pages 1262–1270, Piscataway, NJ, USA, 2010. IEEE Press.

[49] Matthew L. Ginsberg. Counterfactuals. *Artificial Intelligence*, 30(1):35–79, 1986.

[50] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 139–152, New York, NY, USA, 2015. ACM.

[51] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 357–371, New York, NY, USA, 2018. Association for Computing Machinery.

[52] Matt Hamblen. Programmable chips for data center switches catch fire with 20% annual growth. `https://www.fierceelectronics.com/electronics/programmable-chips-for-data-center-switches-/catch-fire-20-annual-growth`, August 2019.

[53] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 71–85, Seattle, WA, April 2014. USENIX Association.

[54] Vipul Harsh, Tong Meng, Kapil Agrawal, and P. Brighten Godfrey. Flock: Accurate network fault localization at scale. `https://arxiv.org/pdf/2305.03348.pdf`, 2023.

[55] Vipul Harsh, Tong Meng, Kapil Agrawal, and Philip Brighten Godfrey. Flock: Accurate network fault localization at scale. *Proc. ACM Netw.*, 1(1), jul 2023.

[56] Vipul Harsh, Wenxuan Zhou, Sachin Ashok, Radhika Niranjan Mysore, Brighten Godfrey, and Sujata Banerjee. Murphy: Performance diagnosis of distributed cloud applications. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 438–451, New York, NY, USA, 2023. Association for Computing Machinery.

[57] Herodotos Herodotou, Bolin Ding, Shobana Balakrishnan, Geoff Outhred, and Percy Fitter. Scalable near real-time failure localization of data center networks. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 1689–1698, New York, NY, USA, 2014. ACM.

[58] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 73–90, Carlsbad, CA, July 2022. USENIX Association.

[59] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 150–155, New York, NY, USA, 2017. ACM.

[60] Qun Huang, Haifeng Sun, Patrick P. C. Lee, Wei Bai, Feng Zhu, and Yungang Bao. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. SIGCOMM '20, page 404–421, New York, NY, USA, 2020. Association for Computing Machinery.

[61] Vimalkumar Jeyakumar, Omid Madani, Ali Parandeh, Ashutosh Kulshreshtha, Weifei Zeng, and Navindra Yadav. Explainit! – a declarative root-cause analysis engine for time series data. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 333–348, New York, NY, USA, 2019. Association for Computing Machinery.

[62] Srikanth Kandula, Dina Katabi, and Jean-Philippe Vasseur. Shrink: A tool for failure diagnosis in ip networks. In *Proceedings of the 2005 ACM SIGCOMM Workshop on Mining Network Data*, MineNet '05, pages 173–178, New York, NY, USA, 2005. ACM.

[63] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. Detailed diagnosis in enterprise networks. *SIGCOMM Comput. Commun. Rev.*, 39(4):243–254, August 2009.

[64] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 113–126, 2012.

[65] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 15–27, 2013.

[66] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Detection and localization of network black holes. In *Proceedings of the IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*, pages 2180–2188, Washington, DC, USA, 2007. IEEE Computer Society.

[67] Ramana Rao Kompella, Jennifer Yates, Albert Greenberg, and Alex C. Snoeren. Ip fault localization via risk modeling. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 57–70, Berkeley, CA, USA, 2005. USENIX Association.

[68] Stan Z Li. *Markov random field modeling in image analysis*. Springer Science & Business Media, 2009.

[69] Bingzhe Liu, Ali Kheradmand, Matthew Caesar, and P. Brighten Godfrey. Towards verified self-driving infrastructure. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 96–102, New York, NY, USA, 2020. Association for Computing Machinery.

[70] Chang Lou, Peng Huang, and Scott Smith. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 559–574, Santa Clara, CA, February 2020. USENIX Association.

[71] Liang Ma, Ting He, Ananthram Swami, Don Towsley, Kin K. Leung, and Jessica Lowe. Node failure localization via network tomography. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 195–208, New York, NY, USA, 2014. ACM.

[72] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. *ACM SIGCOMM Computer Communication Review*, 41(4):290–301, 2011.

[73] Edgar Costa Molero, Stefano Vissicchio, and Laurent Vanbever. Fast in-network gray failure detection for isps. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 677–692, New York, NY, USA, 2022. Association for Computing Machinery.

[74] Radhika Niranjan Mysore, Ratul Mahajan, Amin Vahdat, and George Varghese. Gestalt: Fast, unified fault localization for networked systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 255–267, Philadelphia, PA, 2014. USENIX Association.

[75] Pooria Namyar, Behnaz Arzani, Daniel Crankshaw, Daniel S. Berger, Kevin Hsieh, Srikanth Kandula, and Ramesh Govindan. Mitigating the performance impact of network failures in public clouds, 2023.

[76] P4.org Applications Working Group. In-band Network Telemetry (INT) Dataplane Specification Version 2.1. `https://github.com/p4lang/p4-applications/blob/master/docs/INT_v2_1.pdf`, 2020.

[77] Venkata N. Padmanabhan, Lili Qiu, and Helen J. Wang. Passive network tomography using bayesian inference. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment*, IMW '02, page 93–94, New York, NY, USA, 2002. Association for Computing Machinery.

[78] Palo Alto Networks. Critical Issues Addressed in PAN-OS Releases. `https://knowledgebase.paloaltonetworks.com/KCSArticleDetail?id=kA10g000000Cm68CAC`, 2020.

[79] Yanghua Peng, Ji Yang, Chuan Wu, Chuanxiong Guo, Chengchen Hu, and Zongpeng Li. detector: a topology-aware monitoring system for data center networks. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 55–68, Santa Clara, CA, 2017. USENIX Association.

[80] Matthew E. Peters, Sebastian Ruder, and Noah A. Smith. To tune or not to tune? adapting pretrained representations to diverse tasks. In *Proceedings of the 4th Workshop on Representation Learning for NLP (RepL4NLP-2019)*, pages 7–14, Florence, Italy, August 2019. Association for Computational Linguistics.

[81] Rahul Potharaju and Navendu Jain. Demystifying the dark side of the middle: a field study of middlebox failures in datacenters. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, page 9–22, New York, NY, USA, 2013. Association for Computing Machinery.

[82] Arjun Roy, Deepak Bansal, David Brumley, Harish Kumar Chandrappa, Parag Sharma, Rishabh Tewari, Behnaz Arzani, and Alex C. Snoeren. Cloud datacenter sdn monitoring: Experiences and challenges. In *Proceedings of the Internet Measurement Conference 2018*, IMC '18, page 464–470, New York, NY, USA, 2018. Association for Computing Machinery.

[83] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C. Snoeren. Passive realtime datacenter fault detection and localization. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 595–612, Boston, MA, March 2017. USENIX Association.

[84] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

[85] Vishwanath Seshagiri, Darby Huye, Lan Liu, Avani Wildani, and Raja R. Sambasivan. Identifying Mismatches Between Microservice Testbeds and Industrial Perceptions of Microservices. *Journal of Systems Research*, 2(1), June 2022.

[86] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[87] SolarWinds. Configure polling statistics intervals in the Orion Platform. `https://documentation.solarwinds.com/en/Success_Center/orionplatform/content/core-polling-statistics-intervals-sw1829.htm`, Accessed 2021-01-24.

[88] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with pathdump. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 233–248, Savannah, GA, 2016. USENIX Association.

[89] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with switchpointer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, SIGCOMM '20, pages 453–456, Renton, WA, 2018. USENIX Association.

[90] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. Netbouncer: Active device and link failure localization in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 599–614, Boston, MA, 2019. USENIX Association.

[91] VMware. Possible data corruption after a Windows 2012 virtual machine network transfer. `https://kb.vmware.com/s/article/2058692`, 2017.

[92] VMware. Network timeouts or packet drops with VMware Tools 11.x with Guest Introspection Driver on ESXi 6.5/6.7. `https://kb.vmware.com/s/article/79185`, 2021.

[93] Xin Wu, Daniel Turner, Chao-Chih Chen, David A. Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: Automating datacenter network failure mitigation. *SIGCOMM Comput. Commun. Rev.*, 42(4):419–430, aug 2012.

[94] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. Zeno: Diagnosing performance problems with temporal provenance. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 395–420, Boston, MA, February 2019. USENIX Association.

[95] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing missing events in distributed systems with negative provenance. In

Proceedings of the 2014 ACM Conference on SIGCOMM, SIGCOMM '14, page 383–394, New York, NY, USA, 2014. Association for Computing Machinery.

[96] Minlan Yu, Albert Greenberg, Dave Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. Profiling network performance for multi-tier data center applications. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 57–70, USA, 2011. USENIX Association.

[97] Hongyi Zeng, Ratul Mahajan, Nick McKeown, George Varghese, Lihua Yuan, and Ming Zhang. Measuring and troubleshooting large operational multipath networks with gray box testing. Technical Report MSR-TR-2015-55, June 2015.

[98] Qiao Zhang, Guo Yu, Chuanxiong Guo, Yingnong Dang, Nick Swanson, Xinsheng Yang, Randolph Yao, Murali Chintalapati, Arvind Krishnamurthy, and Thomas Anderson. Deepview: Virtual disk failure diagnosis and pattern detection for azure. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 519–532, Renton, WA, April 2018. USENIX Association.

[99] Shenglin Zhang, Ying Liu, Weibin Meng, Zhiling Luo, Jiahao Bu, Sen Yang, Peixian Liang, Dan Pei, Jun Xu, Yuzhi Zhang, et al. Prefix: Switch failure prediction in datacenter networks. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(1):1–29, 2018.

[100] Y. Zhang, M. Brady, and S. Smith. Segmentation of brain mr images through a hidden markov random field model and the expectation-maximization algorithm. *IEEE Transactions on Medical Imaging*, 20(1):45–57, 2001.

[101] Yao Zhao, Yan Chen, and David Bindel. Towards unbiased end-to-end network diagnosis. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '06, pages 219–230, New York, NY, USA, 2006. ACM.

[102] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. Wcmp: Weighted cost multipathing for improved fairness in data centers. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 5:1–5:14, New York, NY, USA, 2014. ACM.

[103] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. Flow event telemetry on programmable data plane. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 76–89, New York, NY, USA, 2020. Association for Computing Machinery.

[104] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 479–491, New York, NY, USA, 2015. ACM.

[105] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 362–375, New York, NY, USA, 2017. ACM.