MONITORING AND ANALYSIS SYSTEM FOR PERFORMANCE TROUBLESHOOTING IN DATA CENTERS

A Thesis
Presented to
The Academic Faculty

by

Chengwei Wang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
December 2013

MONITORING AND ANALYSIS SYSTEM FOR PERFORMANCE TROUBLESHOOTING IN DATA CENTERS

Approved by:

Professor Karsten Schwan,
Committee Chair
School of Computer Science
Georgia Institute of Technology

Professor Karsten Schwan, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Ling Liu
School of Computer Science
Georgia Institute of Technology

Professor Douglas M. Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Matthew Wolf
School of Computer Science
Georgia Institute of Technology

Dr. Vanish Talwar
HP Labs
Palo Alto, CA

Dr. Mohamed Mansour
Amazon.com
Seattle, WA

Date Approved: October 18th 2013

To Yuan and My Parents

ACKNOWLEDGEMENTS

I am eternally grateful to my advisor, Professor Karsten Schwan, for his relentless mentoring and support during my Ph.D. pursuit. He is my role model who has been teaching me how to face the challenges in research, career, and life. Without his patience, his guidance and the mindset he engraves in my brain through his words and deeds, I could have never come this far.

I deeply appreciate the advices from my thesis committee members: Professor Douglas M. Blough, Professor Ling Liu, Dr. Matthew Wolf, Dr. Vanish Talwar and Dr. Mohamed Mansour. They help me during the various stages of my research. Their truthful critics with appreciation for my effort drive me to keep improving the quality of this dissertation.

I owe a debt of gratitude to my mentors who introduce me to the industry. They are Dr. Canturk Isci from IBM T.J. Watson Research Labs, who is my first mentor in U.S. when I intern in VMware, Dr. Choudur Lakshminarayan and Dr. Krishnamurthy Viswanathan from HP Labs. They grant me hands-on research and engineering experiences and long-time friendship.

I would like to thank research faculty in CERCS Lab, Dr. Greg Eisenhauer and Dr. Ada Gavrilovska. I always feel being technically enlightened while working with Greg in various research projects, and the research discussions with Ada are inspiring. A lot of friends help me through the highs and lows of my Ph.D. adventure. Fang, Hongbo, Shuang, Dani Rayan, Gong and Mukil deserve a special mention.

I am grateful to have my great parents, Zehe Wang and Yuyan Wu. Their unconditional love is and will always be my source of strength. Also I would like to thank my parents-in-law, Xun Yuan and Ping Xu, for their constant warm support for me,

my wife and my kid, without which our foreign life would be much tougher.

Finally, I thank my amazing wife, Yuan. She is my safe house when I am battling the odds. She is my light tower when I am exploring the seas. And after all, she is a beautiful woman, a understanding wife, a caring mother, and at the end of the world, the old friend beside me sharing stories.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

It was not long ago. On Christmas Eve 2012, a war of troubleshooting began in Amazon data centers. It started at 12:24 PM, with an mistaken deletion of the state data of Amazon Elastic Load Balancing Service (ELB for short), which was not realized at that time. The mistake first led to a local issue that a small number of ELB service APIs were affected. In about six minutes, it evolved into a critical one that EC2 customers were significantly affected. One example was that Netflix, which was using hundreds of Amazon ELB services, was experiencing an extensive streaming service outage when many customers could not watch TV shows or movies on Christmas Eve. It took Amazon engineers 5 hours 42 minutes to find the root cause, the mistaken deletion, and another 15 hours and 32 minutes to fully recover the ELB service. The war ended at 8:15 AM the next day and brought the performance troubleshooting in data centers to world's attention. As shown in this Amazon ELB case.Troubleshooting runtime performance issues is crucial in time-sensitive multi-tier cloud services because of their stringent end-to-end timing requirements, but it is also notoriously difficult and time consuming

To address the troubleshooting challenge, this dissertation proposes **VScope**, a flexible monitoring and analysis system for online troubleshooting in data centers. VScope provides primitive operations which data center operators can use to troubleshoot various performance issues. Each operation is essentially a series of monitoring and analysis functions executed on an overlay network. We design a novel software architecture for VScope so that the overlay networks can be generated, executed and terminated automatically, on-demand. From the troubleshooting side,

we design novel anomaly detection algorithms and implement them in VScope. By running anomaly detection algorithms in VScope, data center operators are notified when performance anomalies happen. We also design a graph-based guidance approach, called VFocus, which tracks the interactions among hardware and software components in data centers. VFocus provides primitive operations by which operators can analyze the interactions to find out which components are relevant to the performance issue.

VScope's capabilities and performance are evaluated on a testbed with over 1000 virtual machines (VMs). Experimental results show that the VScope runtime negligibly perturbs system and application performance, and requires mere seconds to deploy monitoring and analytics functions on over 1000 nodes. This demonstrates VScope's ability to support fast operation and online queries against a comprehensive set of application to system/platform level metrics, and a variety of representative analytics functions. When supporting algorithms with high computation complexity, VScope serves as a 'thin layer' that occupies no more than 5% of their total latency. Further, by using VFocus, VScope can locate problematic VMs that cannot be found via solely application-level monitoring, and in one of the use cases explored in the dissertation, it operates with levels of perturbation of over 400% less than what is seen for brute-force and most sampling-based approaches. We also validate VFocus with real-world data center traces. The experimental results show that VFocus has troubleshooting accuracy of 83% on average.

CHAPTER I

INTRODUCTION

1.1 Troubleshooting in Data Centers: Challenges and Opportunities

In the emerging cloud computing era, enterprise data centers host a plethora of web services and applications, including those for e-Commerce, distributed multimedia, and social networks, which jointly, serve many aspects of our daily lives and business. For such applications, lack of availability, reliability, or responsiveness can lead to extensive losses. For instance, on June 29^{th} 2010, Amazon.com experienced three hours of intermittent performance problems as the normally reliable website took minutes to load items, and searches came back without product links. Customers were also unable to place orders. Based on their 2010 quarterly revenues, such downtime could cost Amazon up to $1.75 million per hour, thus making rapid problem resolution critical to its business. In another serious incident, on July 7^{th}, 2010, DBS bank in Singapore suffered a 7-hour outage which crippled its Internet banking systems, and disrupted other consumer banking services, including automated teller machines, credit card and NETS payments. The cascading failure occurred due to a procedural error while replacing a faulty component in one of the bank's storage systems that was connected to its main computers.

The huge cost of downtime in large-scale distributed systems drives the need for troubleshooting tools that can quickly detect problems and point system administrators to potential solutions. The increasing size and complexity of enterprise applications, coupled with the large scale of data centers in which they operate, make troubleshooting extremely challenging. Problems can arise due to a large variety

of root-causes because of the complex interactions between hardware and software systems. The large volume of monitoring data available in these systems can obscure the root-cause of these problems. Lastly, the multi-tier nature of applications composed of entirely different subsystems managed by different teams complicates problem diagnosis. The following characteristics of enterprise applications and data centers challenge performance troubleshooting.

1. Scale. It is not unusual for a data center to have thousands of servers. Large web companies' data centers can have over 1 million servers. In consolidated data centers, each server can host hundreds of VMs, and each VM hosts hundreds of application processes supporting various interacting components of application services. At this scale, performance problems occur frequently, and in response, software in these systems is written to deal with potential sources of problems through built-in error logging and tracing. With large volumes of monitoring/tracing data, troubleshooting is much like finding a needle in a haystack.

2. Complexity. Performance troubleshooting in the massively distributed environment of today's data centers goes beyond what is done in private data centers that only host applications owned by a single company. (1) Applications commonly consist of distributed software components deployed on different machines or even different data center sites. (2) Components may come from different software vendors or open-source developers. (3) Component interactions are complex, not only due to scale, but also because they use built-in resilience methods like those based on replication, quorums, and automatic restart. (4) Different teams of developers may be responsible for the different services or tiers of SOA-based applications; therefore, multiple teams need to work together when something goes wrong with a service. (5) Public clouds, like Amazon EC2 or Google App Engine, experience greater levels of complexity than private data centers because they host applications from a diverse

set of customers. Data-center operators typically have little knowledge about the implementation logic — one reason being to protect customer privacy. Conversely, for security and compliance reasons, application developers may not have direct access to underlying hardware — requiring them to request operations teams to provide them with the relevant log and trace files for troubleshooting. If analysis of one server's log fails to reveal the problem because a different server is responsible for the fault, the manual, tedious and error-prone process must be repeated.

3. Dynamism. A data center is a shared infrastructure, with frequently changing users, and with applications frequently installed/deployed and removed. Workloads vary over time, *i.e.*, *temporally*, and across data center nodes, *i.e.*, *spatially*. These variations are exacerbated by their aforementioned resilience methods. The situation is even worse in virtualized public clouds, where VMs running a large variety of customer applications can be created, migrated and terminated dynamically.

Scale, complexity, and dynamism make it extremely difficult to diagnose data center problems, yet effective performance troubleshooting remains crucial for both data center users and providers, because data center providers seek:

- increased hardware utilization and reduced resource consumption,
- reduced IT costs,
- consolidation with automated management tools, all
- to optimize their investment at cloud scale.

while users demand:

- increased service availability and reliability,
- increased service performance and productivity,
- more proactive response to customer needs,
- less downtime for increased revenue, all with

- appropriate levels of security and data privacy.

1.2 State of the Art

Previous research on monitoring has created scalable methods for real-time data collection and aggregation [52, 193, 179, 138], to support efficient on-line queries that answer questions like 'which machines have CPU utilization above 90%?'[114]. 'Analysis-focused' research has drawn from areas like data mining, machine learning, and statistics to create techniques that assist in or automate problem diagnosis [59, 23, 38, 40, 110], with high accuracy and significantly reduced human intervention. However, while monitoring has been shown feasible at scale and in real-time, analysis is typically performed after a volume of monitoring data has been written to disk-resident logs, or in a central location, which impedes the scalability of on-line monitoring and analysis tasks. Further, due to lack of underlying infrastructure support, analytics often require global data – over time and space – making it difficult to use them on-demand and in real-time. Finally, in modern virtualized utility or cloud computing systems, operators or administrators have limited visibility into the virtual machines running on data center machines. This prevents them from using problem diagnosis methods that require such insight.

Previous troubleshooting solutions are either brute-force [138, 193], i.e. 'everything is monitored all the time' including both application- and system-level events, or statistically sampling [156, 165, 73, 42] a portion of components and/or for a period of time. Brute-fore solutions do not scale for detailed diagnostics which mostly requires debugging level logging or tracing, because of the undue penalties on application performance.

Current sampling solutions are also not flexible enough for aforementioned heterogeneous multi-tier application because they are using a homogeneous and/or random sampling strategy across all the nodes. For instance, Dapper [165] is using a static

1/1000 sampling rate across all the servers. GWP [156] randomly samples a small portion of machines and uses an event-sampling rate pre-determined by the event type at machine-level. Therefore more sophisticated methods are needed to tune sampling rates to capture desired behaviors, and such tuning must be done differently for each behavior, tier, and type of analysis being performed.

1.3 Thesis Statement

A flexible architecture integrating monitoring and analysis can cope with the diversity of performance troubleshooting needs on performance, cost and interference to applications. VScope system realizes this architecture in large scale virtualized data center environment. It integrates novel anomaly detection algorithms and graph-based guidance approach. It is scalable to 1000s of virtual machines. It has better performance, lower overhead and interference than traditional brute-force approaches have.

1.4 Modeling, Systems & Algorithms : A Cohesive Solution

This dissertation addresses challenges of performance troubleshooting in data centers in a series of cohesive research components. Their relationships are illustrated in Figure 1.

The first research component is Monalytics, a novel, flexible architecture integrating monitoring and analytics for data center management. It is a novel software design for two reasons. First, previous solutions are either monitoring or analytics systems, while Monalytics is a stream analysis system design which can process the monitoring data (i.e. analytics), when it is being collected and transmitted (i.e. monitoring). Secondly, Monalytics is flexible in terms of supporting various kinds of topologies for different monitoring and analysis purposes. As a proof of concept, the usefulness of Monlaytics architecture is validated by two use cases. I also studied

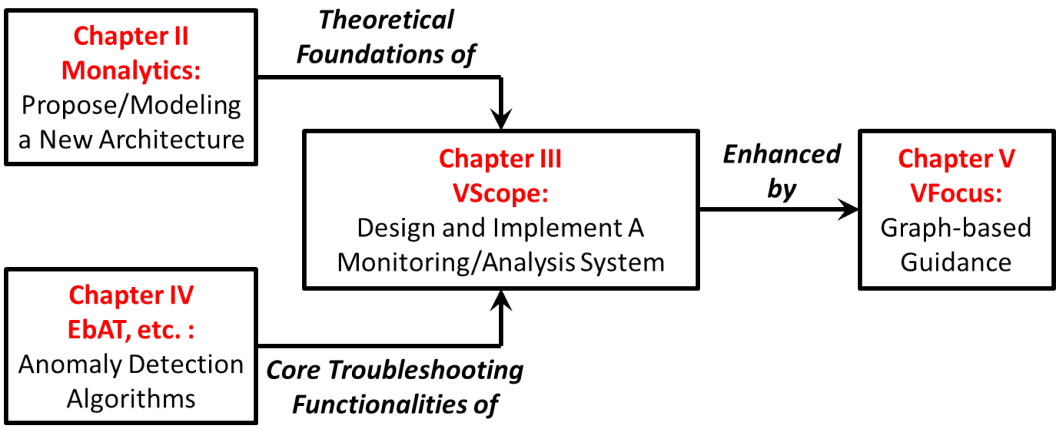

Figure 1: Relationships between research components

the scalability issues of Monalytics architecture at extremely large scale (1 million servers), which concludes that flexibility is in need to build a scalable monitoring and analysis system in large scale data center environment. In sum, Monalytics research builds the theoretical foundation.

VScope inherits the Monalytics concepts with comprehensive system design and implementation. We designed the primitive API for troubleshooting, the automation mechanism for dynamically manipulating monitoring topologies and a set of interaction tracking functionalities to guide the troubleshooting process. VScope is scalable to over 1000 virtual machines. Its usefulness is validated by two use cases in managing large scale big-data applications.

Algorithm research on EbAT (Entropy based Anomaly Tester), and other statistical techniques yields an anomaly detection functionality in VScope system. It is also one of the basic operations of VScope. In this research component, we design and implement a novel anomaly detection algorithm, EbAT and leverages a variety of statistical tools. We evaluate the algorithms in real-world data center traces and compare them with traditional solutions. The results reveal that EbAT considerably outperforms threshold-based solutions..

To further enhance the functionality of VScope, I propose VFocus as a unified approach using graph analysis to track interactions among components in data centers online. We validate, by real-world use cases, that the run-time interaction traces can serve as guidance information for troubleshooting, significantly reducing troubleshooting overheads while achieving high accuracy. We also integrate VFocus into VScope system and test its performance and interference to the data center applications.

1.5 Technical Contributions

In the course of the cohesive research, we make the following technical contributions:

- `A novel flexible software architecture`: we propose and validate Monalytics architecture in use cases which reveals significantly lower interference and higher performance than traditional brute-force approaches.

- `Analytical models for assessing monitoring/analysis systems`: we propose and evaluate analytical models at 1 million nodes' scale. The results confirm that flexible topologies provided by Monalytics architecture can yield considerably better performance than transitional static topologies.

- `A flexible and scalable monitoring/analysis system`: we design and implement VScope at large scale virtualized data center with over 1000 Virtual Machines. VScope is comprehensively tested in that environment. The experimental results show that VScope is scalable, efficient and has low overheads.

- `Novel anomaly detection algorithms`: we design and implement novel algorithms which are further tested in both home-made controlled environment and with real-world data center traces. The results show that our algorithms perform better than traditional approaches.

- `Novel graph-based approach to guide troubleshooting`: we design and implement a novel, graph-based, approach for guiding troubleshooting, called VFocus. We integrate it into VScope and test the method in large scale virtualized data center environment, which shows negligible overhead and high efficiency. VFocus is also validated by real-world data center traces, which shows that VFocus can capture real-world application interactions and can yield good troubleshooting performance.

1.6 Organization of the Dissertation

The remainder of this dissertation is organized as follows:

Chapter 2 describes our work on Monalytics architecture and analytics models.

Chapter 3 presents the design, implementation and evaluation of VScope system.

Chapter 4 describes the design, implementation and evaluation of novel anomaly detection algorithms.

Chapter 5 proposes VFocus, graph-based guidance approach. It presents its design, implementation, usecases and experimental results.

Chapter 6 provides a comprehensive literature study on performance troubleshooting in data centers.

Chapter 7 concludes this research with lessons we have learned, and provides future work directions.

CHAPTER II

MONALYTICS: FLEXIBLE ARCHITECTURE INTEGRATING MONITORING & ANALYTICS

2.1 Introduction

Monitoring and data analysis[1] are two fundamental elements of data center management. Monitoring tracks desired hardware and software metrics. Analysis evaluates these metrics to identify system or application states for troubleshooting, resource provisioning, or other management actions. To effectively manage modern data centers, both monitoring and associated analytics must be performed in real-time and at scales of tens of thousands of heterogeneous nodes with complex network and I/O structures.

Previous research on monitoring has created scalable methods for real-time data collection and aggregation [52, 193, 179, 138], to support efficient on-line queries that answer questions like 'which machines have CPU utilization above 90%?'[114]. 'Analysis-focused' research has drawn from areas like data mining, machine learning, and statistics to create techniques that assist in or automate problem diagnosis [59, 23, 38, 40, 110], with high accuracy and significantly reduced human intervention. However, while monitoring has been shown feasible at scale and in real-time, analysis is typically performed after a volume of monitoring data has been written to disk-resident logs, or in a central location, which impedes the scalability of on-line monitoring and analysis tasks. Further, due to lack of underlying infrastructure support, analytics often require global data – over time and space – making it difficult to use them on-demand and in real-time. Finally, in modern virtualized utility or

[1] We use 'analysis' and 'analytics' interchangeably in this paper.

cloud computing systems, operators or administrators have limited visibility into the virtual machines running on data center machines. This prevents them from using problem diagnosis methods that require such insight.

To address these challenges, we propose a system integrating monitoring with analytics, termed *Monalytics*, which can capture, aggregate, and incrementally analyze data on-demand and in real-time, only where (i.e., in situ) and to the extents needed by intended management actions. This is first introduced in [120], with initial results indicating that Monalytics should be built to respect notions of 'scope' in time and space for (i) acceptable overheads, and (ii) appropriate delays between when certain conditions arise and when they are detected (and thus, can be acted on). To do so, however, requires a flexible architecture to accommodate the changing and diverse characteristics of analytics, with cost-effectiveness in large-scale data centers.

This chapter presents the design and evaluations of Monalytics' flexible architecture built upon dynamic *distributed computation graphs* (DCGs), providing the following technical contributions:

- `Pro re nata (PRN) deployment`: an important property of Monalytics is its instantiations of analytic functions only where and when they are needed. In other words, Monalytics must have capabilities for dynamically *zooming in* to 'interesting' locations and periods of time. Such capabilities can also benefit scalability by substantially reduced costs compared with systems forced to 'watch everything all the time'. *We validate the PRN deployment in two realistic use cases and compare them with traditional brute-force approaches.*

- `Reducing 'Time to Insight' (TTI) and cost`: a vital metric for assessing the performance of monitoring/analysis actions is *Time to Insight* (TTI) capturing the total delay between the time when 'interesting' events occur and the

time when they are recognized (i.e., after analysis is complete). Using this metric and also assessing the costs incurred along with different values of TTI, *we evaluate the cost-effectiveness of alternative topologies used to construct DCGs, and validate our novel flexible hybrid DCG design.*

The evaluations are based on both experimental results and performance modeling at scale. They show that, as the key to autonomic data center management, the flexible PRN deployment of DCGs enables continuous operation at scale and is cost-effective in attaining TTI required for various analytics functions. Compared with brute force solutions and traditional static topologies, Monalytics yields up to 92% TTI reduction and 86% lower cost.

The remainder of this chapter is organized as follows. Section 2.2 elaborates challenges for integrating monitoring and analysis in large-scale data centers. The two use cases driving our research are presented in Section 5.4. The design of DCG-based Monalytics software is described in Section 2.4, and a series of analytical models assessing DCG topologies is detailed in Section 2.5. Section 2.6 presents experimental and analytical evaluation results, related work appears in Section 2.7. Conclusion and future work are in Section 2.8.

2.2 Problem Statement

Accommodating the Variety of Analytics: A first step towards integrating monitoring and analytics is to recognize that analytics approaches vary widely in terms of computational complexity and implementation (e.g., centralized or distributed, etc.). As an illustration, we list representative analytics approaches for data center management in Table 21[2]. The second column describes core algorithms, including simple sorting or traversal algorithms, machine learning methods (e.g., TAN–Tree

[2] N:number of monitoring samples, n:number of metrics in each monitoring sample, k:number of centroids, Δ:increment number of samples, $\{p, e, S, E, W, \tau, m\}$:approach-specific parameters

Table 1: Typical Analytics Approaches

Name	Core Algorithms	Computational Complexity
MAX [179]	Traversing	O(nN)
Top-K [179]	Sorting	O($nNlogN$)
Signatures [61]	TAN Bayes K-Clustering	O(n^2N) [76] O($N^{nk+1}logN$) [96]
Pinpoint [110]	UPGMA Clustering	O(N^2) [145]
Magpie [40]	Incremental Clustering	O($n(N+\Delta)$) [49]
Pranaali [117]	TAN Bayes	O(n^2N) [76]
Black-Box [21]	Nesting Algorithm Convolution Algorithm	O(Np) O($em+eSlogS$)
E2EProf [21]	Pathmap	O($E[\frac{W}{\tau}]^2$)
Sherklock [38]	Inference Graph	O(3^m)

Augmented Naive Bayes and K-Clustering), and algorithms used for specific purposes, such as Convolution [23] and Pathmap [21]. The computational complexities of these various analytics approaches range from linear to exponential, with differences in their computation styles as well. For instance, *MAX* and *Top-K* could be run in a distributed manner using an aggregation tree, while TAN Bayes is processed at a centralized location. Magpie can use incremental clustering as new monitoring data arrives, whereas [23] needs to collect traces for some period of time and do a one-time off-line analysis.

A challenge resulting from this variety is *how to accommodate such a wide range of analytics for autonomic management in large scale data centers?* Previous research does not address this challenge when designing monitoring or aggregation systems, instead focusing optimization efforts on the communication overheads incurred for monitoring, including message volumes and bandwidth consumption, delays in data delivery, etc., using techniques such as in-network processing [134] and source-based filtering [85, 69, 111]. Section 2.6 shows that traditional static system designs cannot easily accommodate analytics functions of varying computation complexities.

Meeting the Varying Requirements in Time and Space: Autonomic management requires *on-demand* monitoring and analytics to diagnose problems and to

understand system behavior, so as to take timely corrective actions to meet service level objectives. The actual analytics used, however, and the level of detail at which they must operate *vary* over time, across the different components or entities being managed, as well as across the multiple levels of abstraction present in complex data center systems. Specifics for this depend on the types of applications or services deployed, system loads, hardware configurations used, desired service level objectives, etc. In any large scale data center, therefore, there will be heterogeneous and non-uniform monitoring and analytics needs over time and space, and these properties also and perhaps, even more so, hold for cloud environments. The elasticity provided by cloud environments allow system deployments to scale-up and scale-down, thereby directly affecting associated analysis structures and requirements.

The resulting challenge for a Monalytics system is: *how to design a system capable of dynamically configuring monitoring and analysis structures to meet the varying requirements in time and space?* Section 2.6 shows that static solutions do not meet the flexibility needed at large scale, whereas our Monalytics design can address the challenge with significant potential performance improvements.

Improving Cost-Effectiveness: With active management of data center hardware and applications becoming increasingly common, it is important to study the costs of management [167, 126, 97]. This should include budgeting both for the capital costs of management hardware resources and for their operational costs (power, cooling, administration, etc). Such budgeting must differentiate costs incurred when management uses *dedicated* hardware (e.g., HP's iLO [7] or IBM's Director [9]) versus when it is *collocated* with the machines being monitored. Intuitively, the former typically provides performance benefits at higher capital costs, whereas the latter has less capital costs but offers reduced performance since it competes for machine resources with the applications being run in the data center.

A resulting challenge is: *how to design a cost-effective system minimizing management cost while yielding the best possible performance?*. As shown in Section 2.6, when considered at scale, high performance for management may incur potentially prohibitive capital cost. Our research addresses this by explicitly modeling the capital costs of management resources. Preliminary results show that Monalytics can outperform static systems at considerably lower capital costs.

2.3 Motivating Use Cases

2.3.1 ZIA: Zoom-In Analysis in Internet Services

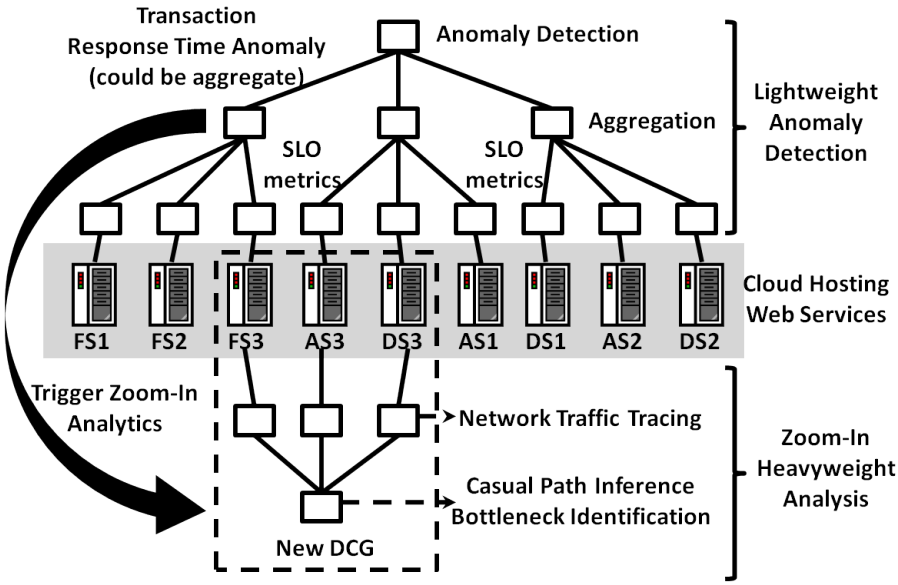

Figure 2: Illustration of ZIA: Zoom-In Analysis in Internet Services

Automatically inferring causal dependencies between components [23, 38, 21] – causal path inference – is known to be useful for identifying performance problems in multi-tier web applications. A typical example is the detection of bottlenecks in the end-to-end latencies experienced by requests. However, the monitoring requirements – large request traces – and high computational complexity make it expensive and most likely, unrealistic to determine all causal paths in large scale data centers, whether they contribute to bottlenecks.

Consider a flexible monitoring and analytics (Monalytics) software that can address this issue by implementing a two-phase approach that evaluates only those dependencies that likely contribute to sudden bottlenecks. This PRN method, termed Zoom-in Analysis (ZIA), allows system administrators to use powerful tools like those required for causal path detection at large scales. Key to the implementation of ZIA is the ability to deploy overlays and analysis functions on any subset of machines, at runtime and as needed (PRN) for further diagnosis.

The phases of ZIA are described with an illustration in Figure 2, showing a practical use case. The group of servers in the middle of Figure (a) exemplifies a utility cloud that hosts Internet services. FS[i], AS[i], DS[i] denote Front-end Server, Application Server, and Database Server for Service i, respectively. The assemblies of rectangles in Figures (a) and (b) represent autonomic deployments of Monalytics overlays.

The first phase is *global light-weight anomaly detection*. Anomaly detection runs continuously with a Monalytics overlay (i.e., the DCG elaborated in Section 2.4) that spans all appropriate data center machines. The overlay collects application-level SLO (Service Level Objective) metrics and system metrics, e.g. transaction response times and CPU utilizations, and it processes them with a low-cost algorithm described in [188, 191]. For simplicity, the figure depicts a hierarchical overlay, but other topologies may also be used. Anomaly detection has each processing unit aggregate and analyze its local metrics and then pass results to its parents. The root has a global view of all machines.

Alarms are raised if anomalies in, say SLOs (e.g., long request duration) are detected, which then triggers the second phase of ZIA. The second phase performs *in-depth analysis in 'problem areas'*, which touches only upon a subset of the machines being monitored, i.e., those for which anomalous behavior has been observed. As illustrated in Figure 2, this involves creating a new overlay at runtime and on-demand

to cover the machines associated with Service 3, Front-end Server 3 (FS3), Application Server 3 (AS3), and Database Server 3 (DS3). This overlay uses a network tracing facility to identify all messages between these servers (RPC requests/replies, IP packages). The data is sent to an analyzer for casual path inference, using an inference engine like the one described in [23]. Knowledge about causal paths can then help identify the potential bottlenecks that caused large end-to-end transaction response times.

The concept of zoom-in analysis generalizes to other problems and applications, and in fact, we next show a second use case in which zoom-in is performed in an entirely different fashion. The effect of using PRN techniques like zoom-in, of course, is that intensive or high overhead analytics can be focused on likely problem areas instead of entire systems. This substantially reduces monitoring overheads and interference to applications compared to a brute force approach that performs causal path inferences all the time.

2.3.2 VMC: Virtual Machine Clustering

A common problem experienced in data centers and utility clouds is lack of knowledge about the mappings of the services being run by or offered to external users to the sets of virtual machines (VMs) that implement them. This makes it difficult to manage VM ensembles – sets of VMs implementing some common service – to attain provider goals like minimizing the resources consumed by certain services or reducing the power they consume on data center machines. A case in point is high network resource consumption when in a public cloud, a VM ensemble running a Hadoop Mapreduce application [4], for instance, is deployed on physical hosts located on different racks rather than on the same rack. This substantially reduces the cross-section bandwidth available in the data center because typically, hardware is configured in ways that offer much less bandwidth across versus within racks. The outcomes are not only

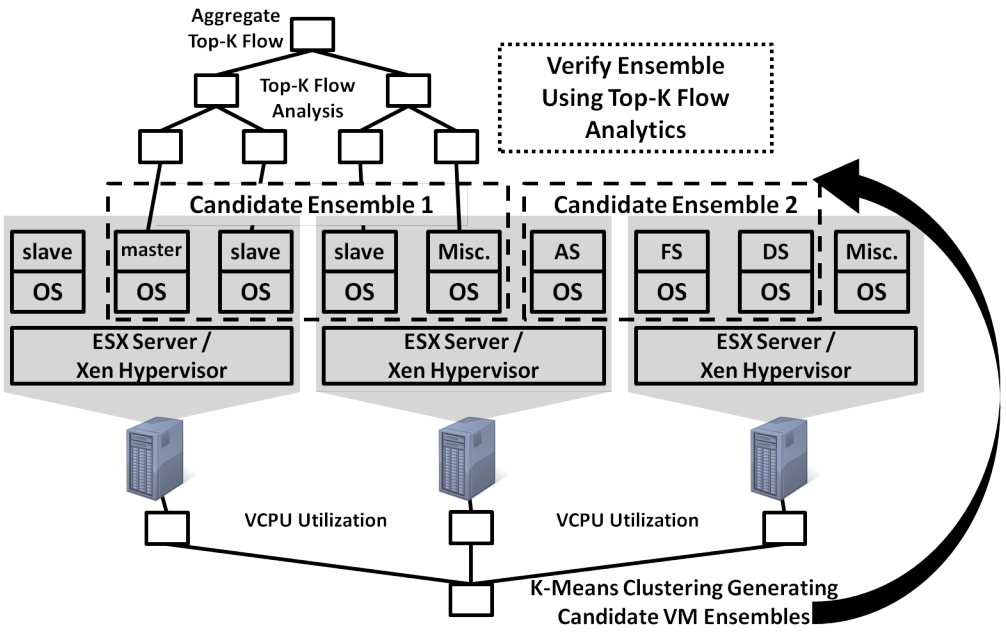

Figure 3: Illustration of VMC: Virtual Machine Clustering

deteriorated performance for the Hadoop application, but also negative impacts on other applications running on these machines. The problem is important because cross-section bandwidth is a limited commodity and increasing it can be costly in terms of the network switches and routers that must be purchased.

Key to properly placing VM ensembles is to first recognize their existence, i.e., to identify them, but this can be difficult. First, system administrators running utility clouds typically have limited knowledge about the applications being run, in contrast to the previous ZIA use case in which we assume an a priori knowledge about the web application's configuration. As a result, administrators must use black-box methods to identify VM ensembles. One such method, based on correlation analysis and described in [94], is efficient in terms of the overheads being experienced, thereby permitting the continuous monitoring necessary to deal with dynamic changes in ensembles and their dynamic arrivals and departures commonly seen in utility data centers. Unfortunately, while overheads are low, the method only discovers *potential* VM ensembles, thus making it necessary to use additional monitoring to distinguish

potential from actual ensembles. With Monalytics, such additional, in-depth analysis can be done where and as needed (PRN) and at runtime. Specifically, the analysis method we use is one that inspects all data exchanged between the VMs in a candidate ensemble, both to confirm the ensemble's existence and to gain additional information for improved ensemble placement, such as inter-VM message volumes. As will be explained in later sections, this PRN method is implemented by installing netfilter modules only into appropriate machines and building an appropriate DCG to analyze the data captured in this fashion (see [94] for additional detail).

A small-scale use case is depicted in Figure 3, each of the three sample servers hosts three virtual machines, where slaves are the worker processes in the mapreduce application, controlled by the master process. Virtual machines running applications other than mapreduce or Internet services are named Misc.. A basic overlay is deployed on three sample servers hosting nine VMs running multiple applications, including a Mapreduce application and a multi-tier web service code. The CPU utilizations of VMs (i.e., VCPU utilizations) are collected on each host. A central node gathering the data runs the lightweight clustering algorithms described in [34]. Its output is a list of potential VM ensembles, i.e., VMs that probably communicate regularly. An ensemble spanning different racks is picked as a potential target for VM migration, but before constructing a migration plan and carrying it out, a PRN method is used to assess the actual amounts of traffic they exchange. This involves creating a new overlay, on-demand, targeting only this ensemble of VMs (a master VM, two slave VM, and a Misc. VM). The new overlay gathers IP package statistics from the VMs and analyzes total network traffic by using *Top-K* flow analysis [115]. The analysis finds the k flows that most contribute to the traffic between any two VMs and their sizes. It eliminates any member of the ensemble with coincidental correlations in terms of CPU usage, and provides the flow data needed to better assess the cost-benefits derived from VM migration.

2.4 System Design

The building block of the flexible Monalytics architecture is the Distributed Computation Graph (DCG), a reconfigurable overlay that undertakes monitoring data collection, exchange, and processing. As illustrated in Figure 4, a DCG is comprised of two types of basic entities, Data Collector(DC) and Monitoring Broker (M-Broker). A DC typically runs on the monitored node, invoking general monitoring tools to collect run time states. Depending on analysis needs, monitoring metrics can be gathered periodically as samples, continuously as traces, or in one-shot. DCs send them to the M-Brokers to which they are attached, using local shared memory when the DCG is collocated with monitored nodes (i.e., the collocated mode) or via the network when the DCG is deployed in dedicated management hardwares (i.e., the dedicated mode). M-Brokers are also connected to each other through a topology for cooperative analysis.

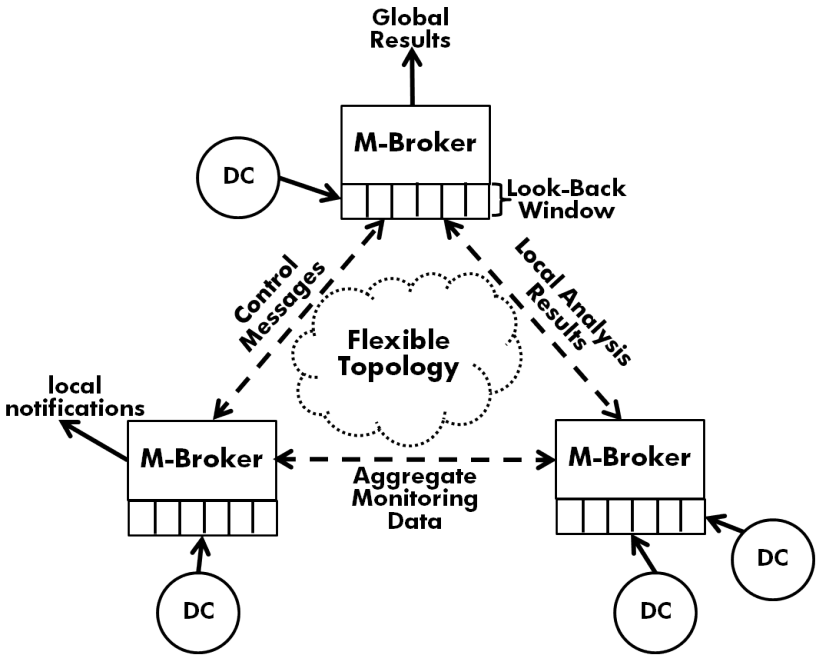

Figure 4: A sample DCG with three M-Brokers and four data collectors (DCs).

A DCG achieves the flexibility needed for scalable, autonomic management

through three design features:

Flexible Analytics Containers (M-Brokers): An M-Broker is the key analytics processing unit in Monalytics. It aggregates the raw data, and passes aggregate results to other M-Brokers for further aggregation for larger scope, or into global states, or for cooperative analysis. Specifically, current M-Brokers maintain analysis state structured as Look-Back Windows (LBWs) supporting on-line analysis. They use these windows to store the monitoring data, to aggregate data, or to maintain intermediate analytics results for some period of time. Analysis actions operate on LBWs, and they are updated as new data flows in. A M-Broker can be deployed in collocation with the node monitored or on dedicated management components such as management blades and processors in the data center. As an analytics container, an M-Broker is able to hold various kinds of analytics functions with associated DCGs. In other words, multiple DCGs implementing different analytics may be constructed by the same set of M-Brokers and DCs.

On-Demand DCG Creation (DCG Controllers): An important attribute of Monalytics is that DCs, M-Brokers, and DCGs are dynamic entities which can be created, connected to each other, and terminated on-the-fly. It is in this fashion that new analytics functions can be initiated and stopped on demand, and existing functions can be adjusted. This on-demand creation and management is done by *DCG controllers*. A controller can create M-Brokers or DCs on any node to which it has access, or reuse existing ones. After that, it connects DCs and M-Brokers using a DCG topology that could be predefined by users or auto-generated based on given management policies. As soon as the DCG is constructed, the DC will start pushing monitoring data into DCG and M-Brokers will process it. The DCG can be terminated via signals issued by the controller, or that task can be assigned to some M-Broker. The controller also tracks the states of DCGs.

For example, in the ZIA use case described in Section 2.3.1, we initially create DCs

on each host to collect application-level monitoring data through JMX and system level metrics using Syststat. Those DCs are attached to M-Brokers that are created and constructed into a hierarchy. Each M-Broker is equipped with the EbAT [188, 191] functions for anomaly detection. When an anomaly is caught, a new DCG is created with DCs on the suspicious nodes and a single M-Broker. The formers trace network flows and relay the data to the latter, which uses a causal path inference engine [23] to analyze the data. When the causal inference process finishes, the DCG is terminated, and the associated DCs and M-Broker can be reclaimed.

Flexible Topologies: The third aspect of the DCG is its ability to use multiple topologies across M-Brokers to meet various analytics requirements in the data center, and to also meet various performance and cost needs. These topologies include the traditional ones used in previous monitoring/aggregation systems. In addition, we propose what we term *hybrid DCGs* that consist of heterogeneous topology structures adapted to different regions of the data center, and interconnected through an inter-region topology. Each region can have its own local topology among the M-Brokers, and the leader M-Brokers of respective regions can be interconnected with another topology. This is very effective at large scale, where the use of a single, static topology to implement various analytics characteristics has substantial negative effects on performance/cost, as shown in Section 2.6. The hybrid approach also makes it easier for the analytics system to scale-up and scale-down, and to support multiple heterogeneous services and data center structures. In addition, it lends itself to scalability of DCG controllers, as shown in Figure 5. At larger scale, a federation of controllers can be used, where each controller manages a region of the DCG. Finally, we note that DCGs and their regions need not correspond to 'physical zones' in the data center. For example, there could be a 1:1 mapping of DCG regions to zones, or there could be multiple DCG regions within a zone, or a DCG region could span multiple zones if zones are small or if large applications run across all of them.

Figure 5 depicts a typical initial deployment of Monalytics, where there are four DCG regions assigned onto 4 racks, respectively. Each region has a *leader* talking to leaders from other regions. This is a hybrid DCG because each region can deploy a different topology, and the topology between leaders can be arbitrary, as well. There is one DCG controller for each rack; they have access to the M-Brokers in their own rack; and they cooperate with each other to jointly manage DCGs. Our current Monalytics prototype implements one controller for a single region; the development of federation support for controllers is in progress.

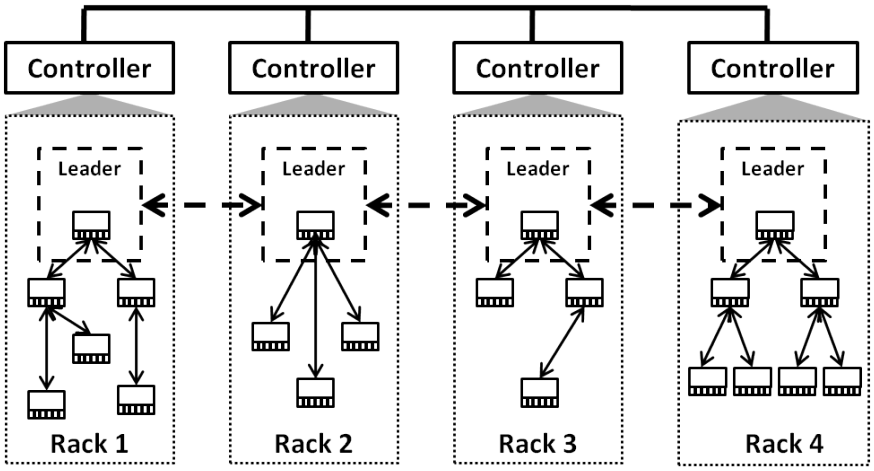

Figure 5: A typical Monalytics deployment on 4 racks

2.5 Modeling DCG Topologies

This section presents a systematic modeling approach to understand Monalytics at scale. These models provide rational estimations and compare the various topologies that Monalytics can create in large scale (from 1000 to 1 million nodes). They are based on real world parameters (network bandwidth, latency, number of nodes, etc.) seen in commercial data centers, and evaluated in realistic configurations, e.g., scale, region size, run times of functions.

Models serve as a sound foundation for the elements of flexibility part of the DCG design. Further, the results obtained from their use reveal new insights on combining

Table 2: Parameters

Parameter	Notation	Example value
Number of Nodes	N	$100 - 10^6$[66]
Network Latency	l	0.25ms[66]
Bisection Bandwidth	B	1 Gbps[25]
Bandwidth Budget (in %)	b	0.1% - 1%
Size of Monitoring Data per Node	s	100KBytes
Number of Metrics	n	1890
Processing Time per Metric	a	$3.5 * 10^{-8}$seconds
Capital Cost per Server	c	$1000[14]
Fraction of Mgmt Cost Per Server	α	1/16[14]
Region Size	N_r	1000

monitoring and analytics, including validation of Monalytics' autonomic features like the PRN methods described earlier.

2.5.1 Traditional Topologies and Hybrid DCGs

Topologies of previous monitoring/aggregation works can be generalized into three types: *centralized, hierarchical tree and binomial swap forest.* In a centralized topology, monitoring data is collected on each node but sent to a central node for analysis. Most of the analysis systems and small to moderate monitoring systems use this approach. Hierarchical trees [193, 146] are widely used in monitoring and aggregation systems, where nodes are organized into a balanced tree (or balanced forest where each tree is according to a different set of attributes), usually with some moderate fanout factor, e.g., 16 [146]. BSF is proposed in [52]. Nodes exchange monitoring data with each other in order and the last two swapping nodes yield the global aggregate data.

Monalytics is capable of creating any of those traditional topologies and in addition, the hybrid DCGs that have significantly higher cost effectiveness in large scale systems, compared to traditional topologies.

2.5.2 Assumptions

We model the TTI of each topology and the associated management cost. For generality, TTI is defined as the latency between when one monitoring sample (indicating event of interests) is collected on each node and when the analysis on all of those monitoring samples has completed. Management cost is modeled as the *capital cost for management hardware and associated software*, i.e., the dollar amount for purchasing management hardware/software resources. We study topologies with dedicated mode and collocated mode in large scale. To the best of our knowledge, our work is the first to model capital cost for management infrastructures in large scale data centers.

Models are based on real word parameters in data centers, as listed in Table 2. The bandwidth resource for each M-Broker in collocated mode is estimated as the product of the bisection bandwidth B and the bandwidth budget b. This estimate reflects the common fact that in many modern data centers, the applications and the monitoring overlay share the same network. It extends on previous models [52] that assume full bandwidth to be available to content aggregation overlays. Further, since monitoring is continuous, which means that it continuously uses the network resource, its bandwidth consumption should be a small fraction of the total bandwidth available in order to confine its interference with applications. Similarly, it is intuitive that on each node, a small portion of resource is used for monitoring/analysis, as captured by a fraction α of its capital cost.

We assume DCs are reporting one monitoring sample at a time. Each sample has a size s due to its use of some number of metrics n. We estimate an intermediate/aggregation result has size s and n metrics as well [3]. We also assume there is support for buffering of raw and aggregated monitoring data throughout the data

[3] In extreme cases, the intermediate result can be much larger than s, without aggregation, or much smaller, with higher 'compression' effect. We believe it is reasonable to pick a value in the middle as an estimation.

center, and that there is enough bandwidth provisioned for monitoring for both the dedicated and collocated strategies. M-Brokers failures and associated costs are beyond the scope of this paper.

2.5.3 Traditional Topologies

Centralized: In a centralized topology, monitoring data collected from each node is sent to a centralized server for analysis. The TTI, noted $T_C(N)$, consists of data delivery time and data processing time. When the aggregate bandwidth of the nodes is smaller than the maximum bandwidth of the central server, the data delivery time is $\frac{s}{B*b} + l$, otherwise it becomes $\frac{s*N}{B} + l$ (central node has full bandwidth because it is dedicated). The processing time is formulated as a function *F(n, M)*, where n is the number of metrics to monitor, and M is the number of instances of each metric. In our analytical models, we assume each node collects n metrics, and uses one instance per metric for analysis, for generality purpose. We can set M to other value representing a different number of instances for analysis, and this will not affect the comparison results shown in Section 2.6. Therefore, the TTI for centralized DCG is:

$$T_C(N) = \frac{s}{B*b} + l + F(n,N), if\, B > B*b*N$$
$$or = \frac{s*N}{B} + l + F(n,N), if\, B \leq B*b*N$$

The management cost for the centralized topology has two parts. The first part is the dedicated central server, with cost of C, and the second part is the fraction of management cost on each node. According to empirical experience [14], there is usually one additional dedicated management server for every 1000 nodes added to the data center. Therefore, the cost measurement $C_C(N)$ is:

$$C_C(N) = \lceil \frac{N}{1000} \rceil * c + N * c * \alpha$$

Hierarchical Tree (HT): In a hierarchical tree, internal nodes have similar fanout of at most d, and the leaf nodes have depth of at most $\lceil \log_d N \rceil$ for a system of N

nodes. Starting from the leaf level, the nodes at each level can be divided into groups with at most d members. A group can be treated as a centralized topology where children report to their parent. The groups at the same level process data in parallel. Hence, the processing time is $\frac{d*s}{B} + l + F(n, d)$. For the top single group level with the root as the central server, the processing time is $\frac{\lceil \frac{N}{d^{\lceil \log_d N \rceil - 1}} \rceil * s}{B} + l + F(n, \lceil \frac{N}{d^{\lceil \log_d N \rceil - 1}} \rceil)$. A parent analyzes its children's data and sends the results to the next parent at the higher level. This data flow between levels is sequential. Hence, the TTI in dedicated mode is:

$$\begin{aligned} T_{HT}(N) =& (\lceil \log_d N \rceil - 1) * (\frac{d * s}{B} + l + F(n, d)) + \\ & \frac{\lceil \frac{N}{d^{\lceil \log_d N \rceil - 1}} \rceil * s}{B} + l + F(n, \lceil \frac{N}{d^{\lceil \log_d N \rceil - 1}} \rceil) \end{aligned}$$

If HT is collocated with the monitored system, the bandwidth resource for each node for aggregation is limited. Therefore, the TTI $\bar{T}_{HT}(N)$ is:

$$\begin{aligned} \bar{T}_{HT}(N) =& (\lceil \log_d N \rceil - 1) * (\frac{d * s}{B * b} + l + F(n, d)) + \\ & \frac{\lceil \frac{N}{d^{\lceil \log_d N \rceil - 1}} \rceil * s}{B * b} + l + F(n, \lceil \frac{N}{d^{\lceil \log_d N \rceil - 1}} \rceil) \end{aligned}$$

The management cost for dedicated mode $C_{HT}(N)$ is the total cost of its internal nodes, of the root, and the inherent cost on each node. The number of parents with N leaves is $\sum_{i=1}^{\lceil \log_d N \rceil} \lceil \frac{N}{d^i} \rceil$:

$$C_{HT}(N) = \sum_{i=1}^{\lceil \log_d N \rceil} \lceil \frac{N}{d^i} \rceil * c + N * c * \alpha$$

In collocated mode, the only cost for management is the fractional management cost on each node:

$$\bar{C}_{HT}(N) = N * c * \alpha$$

Binomial Swap Forest (BSF): In a Binomial Swap Forest (BSF) [52] topology, each node computes an intermediate result by repeatedly swapping (exchanging) data

with other nodes. Two nodes swap data by sending to each other the intermediate results they have so far, letting each to compute the new results of both nodes' data. The swaps are organized so that a node only swaps with one other node at a time, and each swap roughly doubles the number of nodes whose data are processed a node's intermediate result, so that the nodes will compute the global result in roughly log(N) swaps. The sequence of swaps performed by a particular node form a binomial tree with that node at the root. BSF runs in collocated mode and its TTI and cost are:

$$T_{BSF}(N) = \lceil \log_2 N \rceil * (\frac{s}{B * b} + l + F(n, 2))$$
$$C_{BSF}(N) = N * c * \alpha$$

2.5.4 Hybrid DCGs

In hybrid DCGs, any of the traditional topologies described above (centralized, hierarchy, BSF) can be used locally within regions, and also to interconnect region leaders. The resulting graph is thus a hybrid of multiple traditional topologies, same or different, interconnected together. For a hybrid DCG that consists of m *regions* which process their local monitoring data in parallel, the time for leaders to receive the aggregated data would be $\max_{i=1}^{m} T_{INTRA}(R_i)$, where $T_{INTRA}(R_i)$ is the TTI for region i, and R_i is the number of nodes in region i. The region leaders then cooperate to perform the next level processing in a time of $T_{INTER}(m)$. Therefore, the TTI and capital cost of a hybrid DCG are formulated as follows:

$$T_{HYBRID}(N) = \max_{i=1}^{m} T_{INTRA}(R_i) + T_{INTER}(m)$$
$$C_{HYBRID}(N) = \sum_{i=1}^{m} C_{INTRA}(R_i) + C_{INTER}(m)$$
$$(N = \sum_{i=1}^{m} R_i)$$

In practice, there could be numerous topology combinations for a hybrid DCG. For purposes of modeling and evaluation, we pick three representative ones: (1)

Centralized-BSF Monalytics (CB) uses a centralized topology within each region and a BSF topology inter-region; (2) *Centralized-Hierarchy-BSF Monalytics* (CHB) has centralized or hierarchical topologies within regions and a BSF topology to connect these regions; and (3) *BSF-BSF Monalytics* (BB) uses a BSF topology both intra- and inter-region, and is also 'hybrid' in the sense that the sizes of the intra-region BSF and the inter-region BSF topologies are different. The formulation for the TTI and capital cost for these examples can be obtained by substituting appropriate formulations for centralized, hierarchical, and BSF topologies in the $T_{HYBRID}(N)$ and $C_{HYBRID}(N)$ equations above. For example, assuming all regions have same size N_r for simplicity, the formulations for TTI and capital cost for the *Centralized-BSF* hybrid DCG would be:

$$T_{CB}(N) = T_C(N_r) + T_{BSF}(\lceil \frac{N}{N_r} \rceil)$$
$$C_{CB}(N) = \lceil \frac{N}{N_r} \rceil * C_C(N_r) + C_{BSF}(\lceil \frac{N}{N_r} \rceil)$$

2.6 Experiments and Evaluations

We have implemented a prototype of Monalytics in C/C++ using the EVPath library [69]. Experimental evaluations at smaller scale use a virtualized environment with 36 virtual machines on 12 physical hosts, complemented by a set of large-scale analytical evaluations to up to 1 million nodes. The results support three main conclusions. First, Monalytics' PRN deployment feature results in substantial TTI reduction compared to traditional approaches. Second, Monalytics has significantly lower cost and interferes less with applications, compared to static systems watching everything all the time. Third, the flexible, hybrid DCG design provides promising advantages in terms of performance and cost, compared to static, single topology solutions (up to 92% TTI reduction and 86% lower cost).

2.6.1 Experimental Evaluations

Setup: Our testbed consists of 12 blade servers, each hosting 3 VMs. We realize the two use cases described in Section 5.4, using our Monalytics prototype to monitor/analyze two application benchmarks: (1) RUBiS[53] representing Internet services and (2) Hadoop[4] representing MapReduce applications. We use the 32 out of total 36 VMs as the monitored nodes running RUBiS and Hadoop instances. The remaining 4 VMs are used to emulate user requests to RUBiS and for PRN deployments of new M-Brokers.

The initial DCG is a hierarchical tree with max fanout of 8. The 32 VMs are leaves, each having one DC and one M-Broker deployed. Some of them are also reused as internal M-Brokers as 1 root and 4 parents. Lightweight anomaly detection [188, 191] and on-line clustering [34] approaches are running on the hierarchy continuously and throughout the experiment. As described in Section 5.4, the DCG controller creates new centralized DCGs to run deeper analysis functions, causal path inference or Top-k, on demand. Causal path inference needs to gather and merge a considerable volume of traces collected from 'suspicious' VMs, so the transmission of monitoring data has an important effect both on TTI and on application interference, which we will discuss next. The *Top-K* approach analyzes IP packets locally on each candidate cohort member, so the network overhead is low. However, since it parses every input and output IP packet, its CPU consumption is high, which again, can significantly interfere with the applications, especially when running it on every node all the time.

Experiments measure the TTI, monitoring data volume, and interference with applications. We compared our PRN solution with brute-force solutions that 'turn on' analytics functions all the time or on every node. To be more specific, the brute-force ZIA approach collects network traffic trace from all the nodes involved because it does not have the 'zoom-in' capability. By the same token, brute-force VMC runs *Top-K* functions all the time on a VM ensemble. results show that the PRN and in

situ approaches result in up to 86% TTI reduction and considerably less interference with application performance. The data size needed for Monalytics is also up to 95% smaller than that of brute-force.

Results: Zoom In Analysis In the ZIA use case, the data center hosting multiple Internet services is monitored, and one of those services has performance problems that requires further analysis via casual path inference. We emulate those problems by injecting anomalies described in [188]. In the experiment, each Internet service is a RUBiS instance embedded in 4 VMs running 1 Apache web server, 2 Tomcat application servers, and 1 MySQL database server, respectively. We vary the system scale from 8 VMs (2 services) to 32 VMs (8 services). Monalytics deploys a DCG on the 4 VMs running the problematic service, on-demand and when needed, while the brute force approach triggers data collection and transmission on all of the VMs. The TTI of Monalytics, then, is significantly shorter than that of the brute force approach, as shown in Figure 6. As the scale increases, the TTI gap grows rapidly, indicating even better performance of Monalytics at larger scales, with a 86% TTI reduction on 32 VMs. In addition, the amount of data collected, transmitted, and processed remains low even as the system scales, with the consequent benefits shown in Figure 7, in comparison to the brute force approach in which the data size increases with increased system scale.

Results showing the degree of interference with application performance are encouraging, as well. Here, we measure the average throughput of all RUBiS services as an indication of performance. In Figure 8, the baseline bar is the average throughput without monitoring/analysis deployed. The Monalytics and brute force bars represent the average throughput for RUBiS with continuous monitoring/analysis. We can see that as the scale increases, the brute force approach results in high interference with application, severely reducing their throughput. This is because the brute

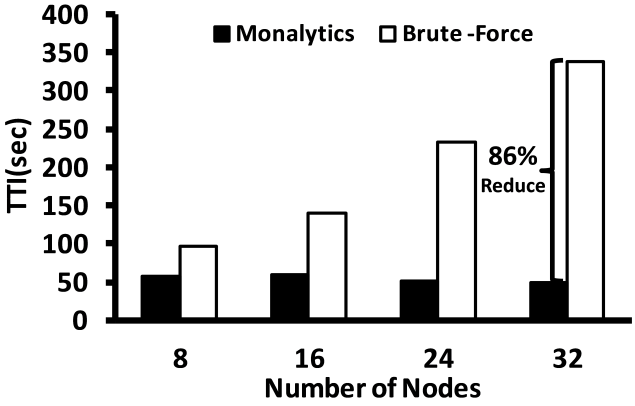

Figure 6: ZIA usecase TTI w.r.t. number of nodes

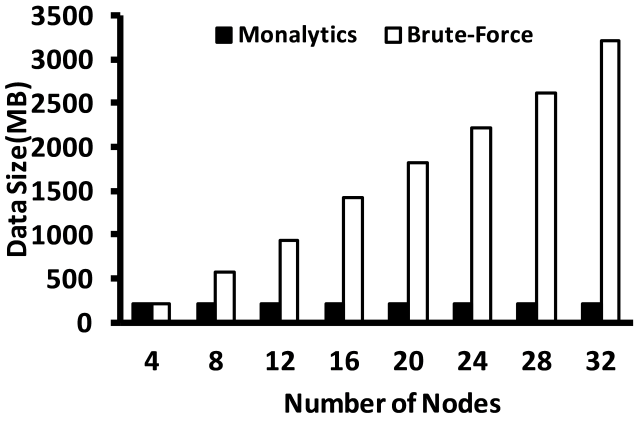

Figure 7: ZIA usecase data size w.r.t number of nodes

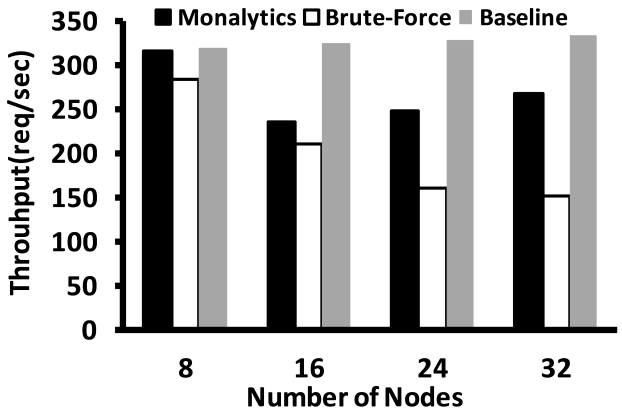

Figure 8: ZIA usecase interference to application

force approach consumes substantial network bandwidth to transmit the larger volumes of monitoring data collected, which in turn slows down the performance of the applications sharing the same network. The Monalytics approach has overall lower interference than the brute force approach (up to 44% higher throughput), and the effect of interference decreases as system scales (due to the increased availability of total network bandwidth).

Results: Virtual Machine Clustering We run a MapReduce application that uses a BBP-type method to compute the exact binary digits of π [4], on 4 VMs to 32 VMs. Figure 9 shows the job completion times when the total workload of the application is fixed, i.e., the workload on each VM is reduced as the system scales. Figure 10 depicts the job completion times when the workload on each slave VM is fixed. In both scenarios, Monalytics incurs much less interferences than the brute force solution, because the former turns on the CPU consuming *Top-K* functions only when needed, whereas the latter runs them all the time, thereby unnecessarily stealing CPU cycles from the Hadoop application. When total workload is fixed, the completion time deceases as the system scales due to the parallel executions on more

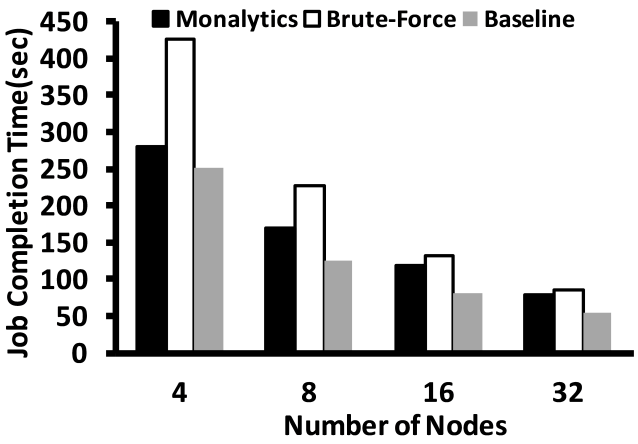

Figure 9: VMC usecase interference, total workload fixed

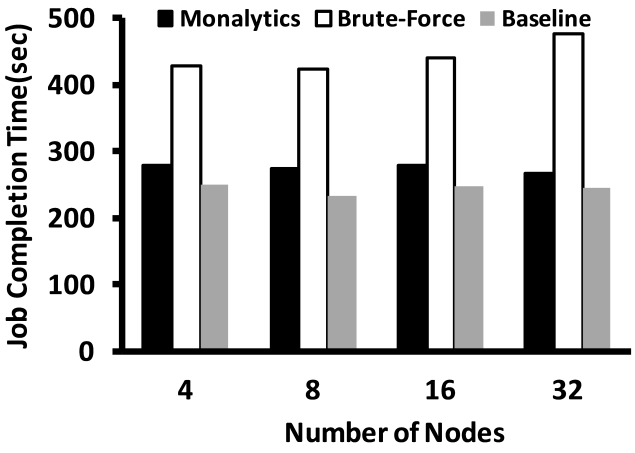

Figure 10: VMC usecase interference, slave workload fixed

Table 3: $F(n, M)$ Functions Representing Two Types of Analytics

Types of Analytics	$F(n, M)$
Linear-Time Approach	$a * n * M$
Quadratic-Time Approach	$a * n^2 * M^2$

Table 4: Abbreviations used for the Monalytics topologies

Centralized	Centralized
Hierarchical Tree (Collocated)	HT
Hierarchical Tree (Dedicated)	HT-Dedicated
Binomial Swap Forest	BSF
Centralized-BSF	CB
BSF-BSF	BB
Centralized-Hierarchy-BSF	CHB

slaves. The baseline bars in Figure 9 reflect this trend. Accordingly, the effects on completion time decrease, because the application's workload can be finished in a few CPU time slots without interruption. When the per slave workload is fixed, the baseline does not change much because slaves run in parallel, and the job completion time is largely determined by the running time of a slave. Because a slave needs to run a relatively longer time, the brute force approach drags down its performance by running the Top-K algorithm throughout the Hadoop execution. Monalytics induces much less interference on job completion time (an average 12% increase) than brute force (85% increase).

2.6.2 Analytical Evaluations at Large Scale

In this section, we evaluate the Monalytics DCG topologies at scale using the models described in Section 2.5.

Parameters and Estimations: The parameter values used are based on real world practices, and are shown in Table 2 (third column). The characteristics of monitoring data were determined using several runs of a microbenchmark consisting of over 15 well known system monitoring tools. This was used to estimate the monitoring data

size and number of metrics per node and per data collection interval[4]. The data processing time on each node, represented by *F(n, M)*, can have various values due to the variety of analytics functions possible. Since it is impossible to exhaust all possibilities, we instantiate *F(n, M)* for our evaluations with two straight-forward and representative run time functions, shown in Table 3[5]. For simplicity, the hybrid DCGs in our evaluations are assumed to have the same region sizes.

Evaluation Results: We compare the TTI and capital cost of various topologies that can be created by Monalytics (see Table 4 for acronyms). The flexibility offered by hybrid DCGs results in better performance with low cost at scale. In particular, Figures 11, 12, and 13 compare the Monalytics' hybrid topologies with traditional HT and BSF topologies when using linear-time and quadratic time analysis functions. As shown in Figure 11, Monalytics CB provides the second shortest TTI for linear-time analysis.

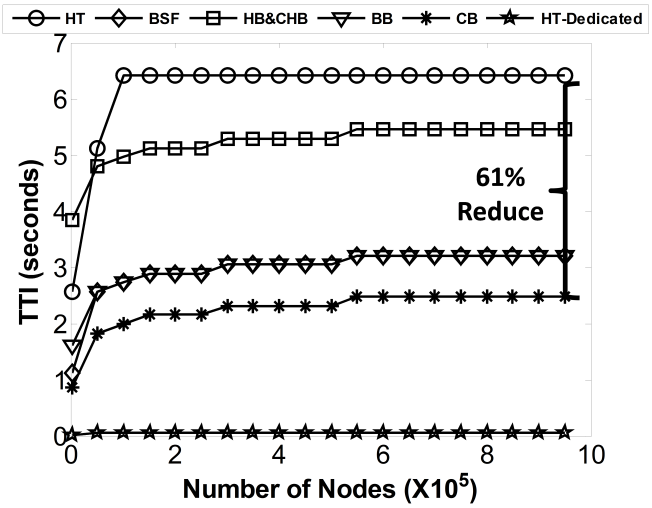

Figure 11: TTI of complexity O(N) w.r.t number of nodes

[4]Note that some previous works [52] have used larger estimates of data sizes, dependent on number of application metrics. We did model evaluations with larger data sizes, as well, and found similar results and conclusions.

[5]a is the processing time of one metric, and $n * M$ is the total number of metrics

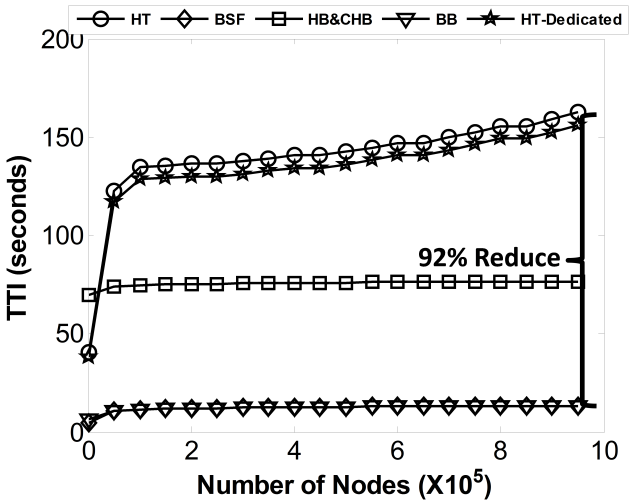

Figure 12: TTI of complexity O(N^2) w.r.t number of nodes

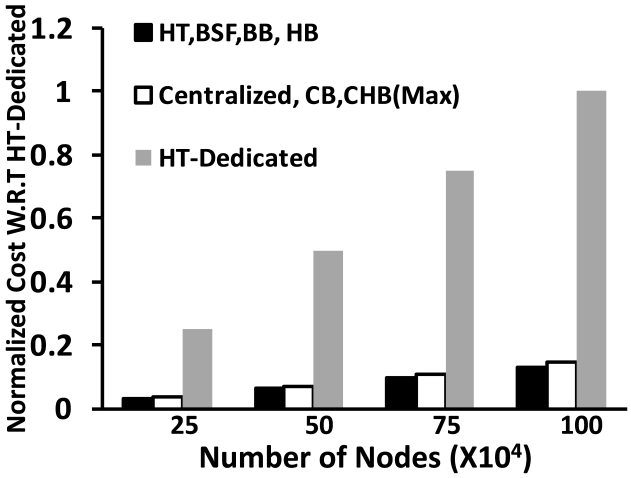

Figure 13: Cost w.r.t number of nodes

At the scale of 1 million nodes, the Monalytics CB exhibits a 61% TTI reduction over collocated HT topology. The dedicated HT has the best TTI but also has the highest capital cost, as depicted in Figure 13(normalized capital cost with respect to HT-Dedicated cost at scale 1 million). Hybrid DCGs have significantly lower cost, over 85%.

For quadratic-time analysis, Figure 12 shows that Monalytics BB yields the best performance, with a significant 92% smaller TTI than that of the dedicated HT. CB, not shown in Figure 12, has the highest TTI(over 200 seconds). The computation at each M-Broker dominates TTI for quadratic-time analysis. Since BSF has much less data to compute per node than what HTs have, BSF and Monalytics BB both yield shorter TTI. Conversely, with linear run time, the communication overhead dominates, so the dedicated HT with larger bandwidth out-performs others.

Other hybrid DCGs can perform comparably better than traditional topologies, as well. The Monalytics CBH has better performance than the collocated-HT in both linear and quadratic runtime, and its maximum cost is slightly higher than that of collocated HT, BSF, BB, and HB, but much lower than dedicated-HT, about 86%.

Hybrid DCGs are more effective due to their ability to mix and match the best topologies that meet local analytics requirements and that also perform best at varied local scale levels. In Figure 14, the completion time of the centralized topology is linear to the scale of data centers, which makes performance prohibitively low at a large scale like 1 million nodes. However, this topology is not always the worst choice because for less than about 2000 nodes, its TTI is less than that of collocated HT and BSF because it has only one level of hierarchy. It also has much lower cost than the dedicated HT, as shown in Figure 13. Hence, using a centralized topology at appropriate scale may yield good performance. This insight is leveraged to build hybrid CB with encouraging cost-effectiveness in Figure 11, because for CB, the intra-region centralized topology has smaller TTI than the collocated HT and BSF when

the number of nodes is within 2000, and inter-zone processing time is the second smallest.

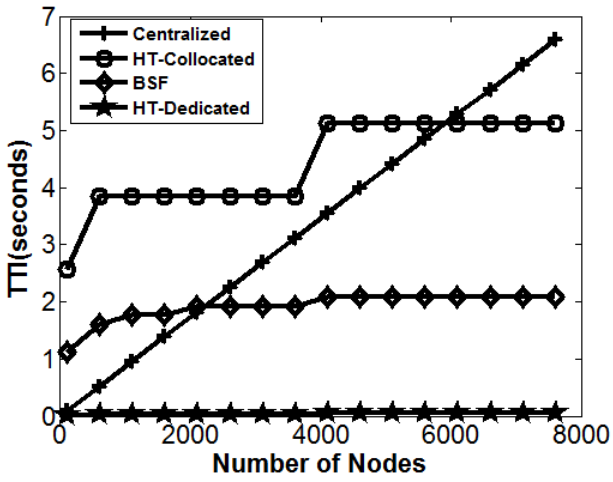

Figure 14: Change of performance ranking at different scales

We also study how topology configurations like fan-out factors affect performance and cost. Figures 15 and 16 reveal that smaller fan-out contributes to smaller TTI. That's because each internal node processes less data with smaller fan-out. Although the height of the tree is larger with smaller fan-outs, the increase has much less effect than that of input data processed by each parent. The improved performance, however, comes with higher cost. As shown in Figure 17, the dedicated HT with smaller fan-out factors have higher management server costs, because the number of internal nodes increases. The results tell us that configuration may have substantial impact on performance, and the trade-off between performance and cost. Flexibility in DCGs can result in on-demand changes in configurations to meet these different trade-off needs.

Finally, evaluations also show that there is hardly a 'one size fits all' topology for all scale and analysis needs. For example, dedicated HT and Monalytics CB which has best performance in linear runtime analysis are among the worst in quadratic time,

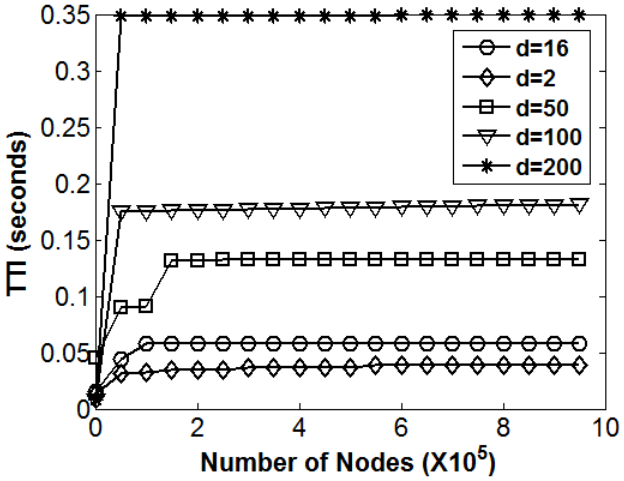

Figure 15: TTI of hierarchy topology O(N)

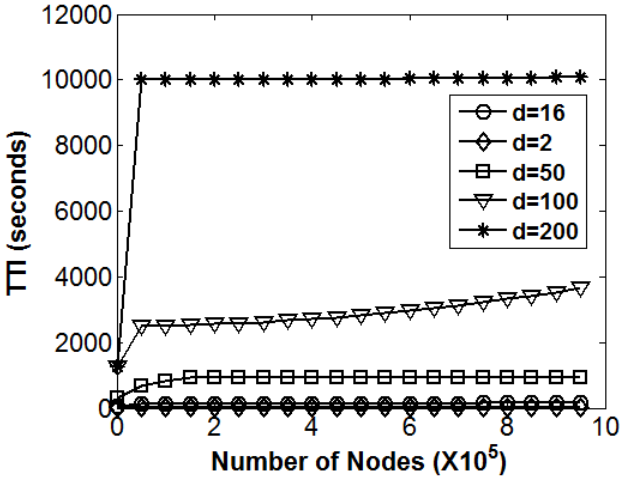

Figure 16: TTI of hierarchy topology O(N^2)

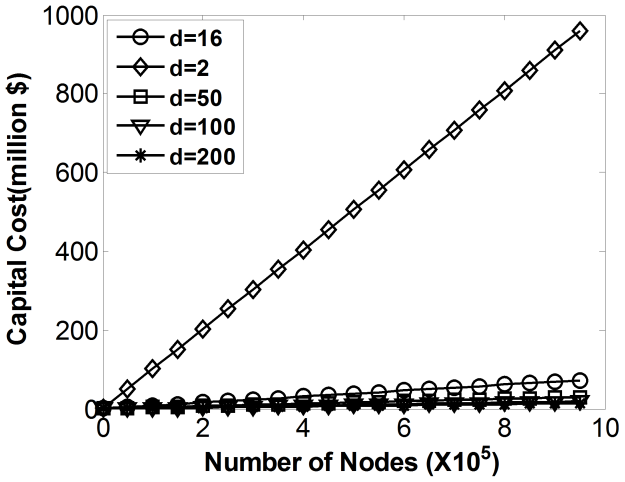

Figure 17: Cost of HT-Dedicated

which suggests that, instead of using a static, single topology, a dynamic hybrid topology should be applied to meet the changing analytics requirements.

2.7 Related Work

`Monitoring systems` [138, 78, 31, 166, 85] are designed to monitor the status of large networked systems or large cluster machines. Some of these leverage hierarchical architectures for scalability. Ganglia [138], in particular, uses multicast messages inside a cluster and federations between clusters. While they are widely used and exhibit high performance in reporting states at large scale, their analysis capabilities are limited.

`Aggregation systems` [134, 52, 193, 179, 146] aggregate distributed data with large volumes, and usually provide mechanisms to query the runtime aggregate states. While Monalytics is similar in its ability to perform in-network, distributed processing, there are two major differences. First, Monalytics provides the flexibility to dynamically create, change, and terminate new topologies to meet varying analysis needs. Second, the analysis functions supported by aggregation systems are typically

limited to those with the specific computational properties [134] suitable for aggregation, while Monalytics is designed to support a variety of analytics without the restrains.

`Analytics solutions` [21, 23, 38, 40, 110] are promising sophisticated analysis of system behaviors with high accuracy and significantly reduced human intervention. However, they rarely support analytics on-demand and at large scale. In addition, they are often application-specific rather than supporting the general class of utility data center applications targeted by our work.

2.8 Conclusions

We presented the Monalytics software architecture for integrating monitoring and analytics in large scale data centers, with flexibility for supporting a variety of analytics functions. We introduce *pro re nata*(PRN) methods and experimental evaluations are carried out with a Monalytics software prototype implemented in small scale data center running three tier enterprise applications and Hadoop codes. Results clearly show the importance of using PRN, along with the ability of the current Monalytics prototype to support the multiple and sophisticated monitoring/analysis functions required by two realistic use cases. We contribute novel analytical formulations modeling DCG's effects on both the performance and the capital costs of monitoring/analysis, with extensive analytical evaluations in large scale.

CHAPTER III

VSCOPE: MIDDLEWARE FOR TROUBLESHOOTING BIG-DATA APPLICATIONS

3.1 Introduction

In the 'big data' era, live data analysis applications are becoming easy to scale, as well as lucrative or even critical to a company's operation. For instance, by continuously analyzing the live number of page views on its products, an e-commerce website can run a dynamic micro-promotion strategy in which when over 3000 customers are looking at a product for over 10 seconds, an extra 20% discount appears on the web page to increase sales. Other mission-critical examples for e-commerce sites are click fraud and spam detection.

The importance of live data analysis is underscored by the recent creation of real-time or 'streaming' big data infrastructures,[1] including Flume, S4, Storm, Chukwa, and others [32, 147, 137, 155, 63, 127, 74], as shown in Table 5. Conceptually, these are based on the well-established paradigm of stream- or event-based processing [82], but their recent attractiveness stems from the fact that they can be easily integrated with other 'big data' infrastructures, such as scalable key-value stores and MapReduce systems, to construct multi-tier platforms spanning thousands of servers or consolidated virtual servers in data centers. A sample platform integrating Flume and other data-intensive systems is depicted in Figure 18. In Flume, *agents* reside in web or application servers, collecting logs and converting them into key-value pairs. *Collectors*

[1] In this chapter we use the term 'real-time' to refer a latency restriction within seconds or hundreds of milliseconds.

receive and aggregate the local results and insert them into HBase, a distributed, scalable key-value store by which users can query the analysis results on-the-fly. HBase consists of *region servers* that are equipped with a memory cache, termed *MemStore*, and a Write Ahead Log (WAL). The data are first written to the WAL and MemStore before being asynchronously persisted to the back-end distributed file system, HDFS, which is typically shared by other data-intensive batch systems, such as Hadoop-based MapReduce codes used for off-line, long-term analyses. Each tier can scale to 1000s of servers or virtual machines.

Table 5: A List of Representative Real-Time Big Data Systems

Name	Developer	Example Application
Chukwa	Apache	Log Collection & Analysis
HOP	Berkeley	Text Processing
Flume	Apache	Impression Analysis
Kafka	LinkedIn	Relevance Analysis
Scribe	Facebook	Log Aggregation
S4	Yahoo!	Real-Time Web Search
Storm	Twitter	Real-Time Tweet Analysis
System S [82]	IBM	Real-Time Stock Analy.

Crucial to maintaining high availability and performance for these multi-tier applications, particularly in light of their stringent end-to-end timing requirements, is *responsive troubleshooting* – a process involving the timely detection and diagnosis of performance issues. Such troubleshooting is notoriously difficult, however, for the following reasons:

- *Holistic vs. tier-specific troubleshooting.* As illustrated in Figure 18, each tier is typically a complex distributed system with its own specialized management component, like the HBase or Flume masters. Developed by different vendors and/or managed by different operation teams, tier-specific management can improve the availability of individual tiers, but is not sufficient for maintaining an entire application's end-to-end performance, a simple reason being that issues

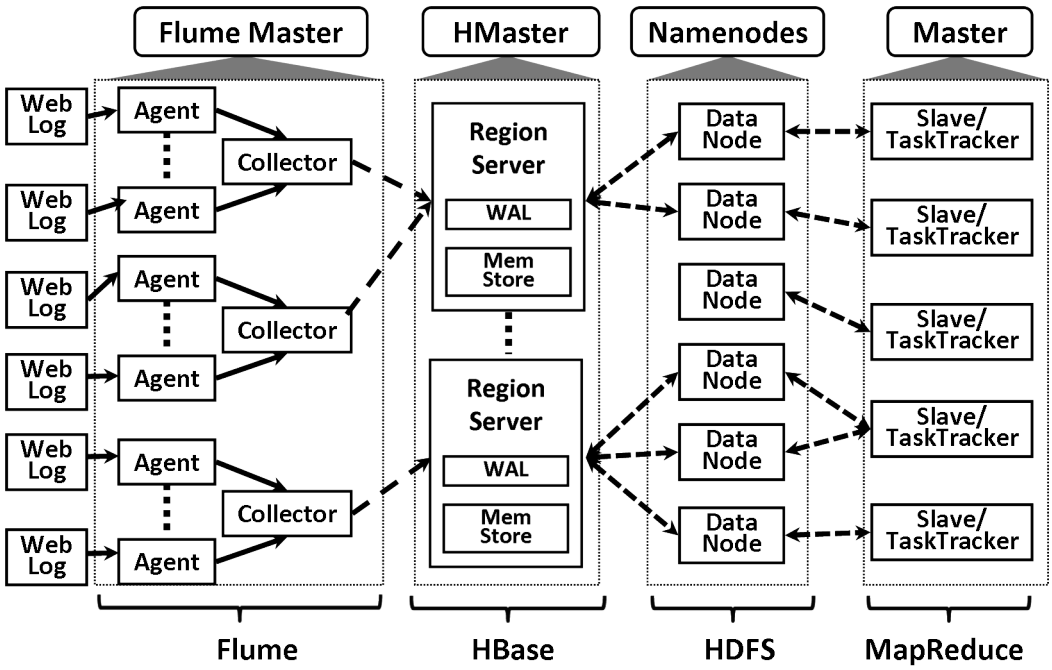

Figure 18: A typical real-time web log analysis application.

visible in one tier may actually be caused by problems located in another. Needed are holistic systems able to efficiently track problems across tiers.

- *Dynamic, per-problem functionality.* Problems in complex, large-scale systems arise dynamically, and for each class of problems, there may be different detection, analysis, and resolution methods. Troubleshooting, therefore, is an inherently dynamic activity, involving on-line capabilities to capture differing metrics and to diagnose/analyze them with potentially problem- and situation-specific methods[187].

- *Scalable, responsive problem resolution.* In latency-sensitive applications like the one in Figure 18, to maintain desired timing, troubleshooting must be conducted both with low perturbation and with high responsiveness: issues must be detected, diagnosed, and repaired without missing too many events and while maintaining availability for other ongoing actions.

- *System-level effects.* Holistic troubleshooting must extend beyond a single application, to also identify the system-level bottlenecks that can arise in today's consolidated data center or cloud computing systems.

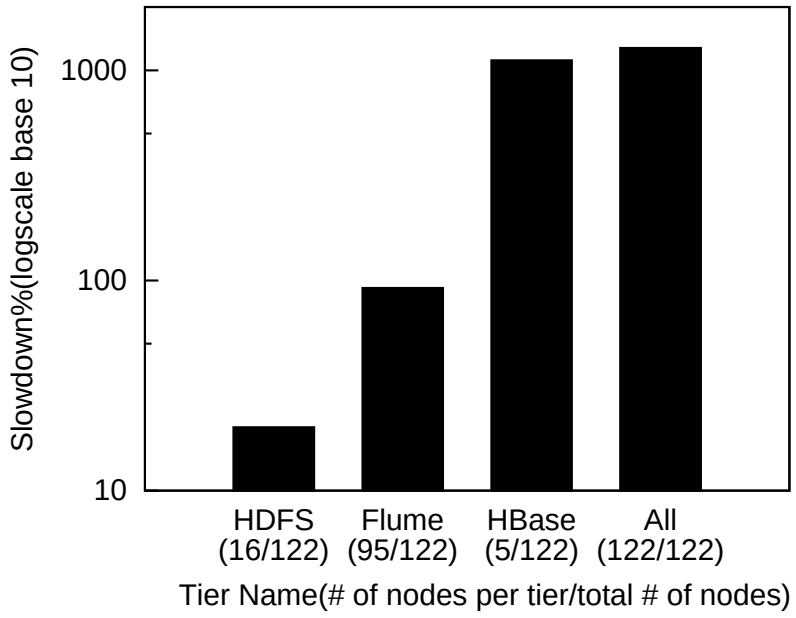

Figure 19: E2E performance slowdown caused by debug-level logging.

Previous troubleshooting systems have not addressed all of these challenges. Solutions that monitor 'everything all the time' [138, 193, 78], including both application- and system-level events, do not scale for detailed diagnostics via say, debug-level logging or tracing with consequent high levels of perturbation. This is shown in Figure 19, where continuously logging application-level debugging events on all of its nodes slows down an application's performance by more than 10 times over the baseline. Sampling [156, 165, 73, 42] for some of the components and/or for some period of time may not only miss important events, affecting troubleshooting effectiveness, but will also bring about serious performance issues when using a homogeneous and/or random sampling strategy across all nodes, e.g., with Dapper [165]'s use of a uniform, low (1/1000) sampling rate. In Figure 19, debug-level logging in the Flume application's HBase tier, the smallest portion of the system (5/122 VMs), results in over

10 times slowdown, which is more than an order of magnitude of the perturbation imposed by debug-level logging in the Flume tier, which has the majority of nodes (95/122). Thus, it is inadvisable to use a high sampling rate for the HBase tier, whereas such a strategy for the Flume tier will likely lead to only modest additional perturbation. An alternative troubleshooting approach chosen by GWP [156] is to randomly pick some set of machines, this may work well if that set is in the HDFS tier, but will be prohibitively costly if the HBase tier is picked. Other approaches such as those taken by Fay [73] and Chopstix [42] to set sampling rates based on the event population still remains unaware of application level perturbation, resulting in the same issue as the one faced by GWP. We, therefore, conclude that *a more flexible system is needed for efficient troubleshooting, where methods can differ for each behavior, tier, and type of analysis being performed.*

The VScope middleware presented in this chapter makes it possible (1) to adjust and tune troubleshooting dynamically – at runtime – for individual tiers and across tiers, (2) to dynamically deploy any analysis action(s) needed to understand the metric data being captured in the ways required by such troubleshooting, and (3) to do so in ways that meet the perturbation/overhead requirements of target applications. To achieve those ends, VScope, as a flexible monitoring and analysis system, offers the following novel abstractions and mechanisms for troubleshooting latency-sensitive, multi-tier data center applications:

1. *Dynamic Watch, Scope, and Query.* VScope abstracts troubleshooting as a process involving repeated *Watch*, *Scope*, and *Query* operations. Respectively, these (i) detect performance anomalies, (ii) 'zoom-in' to candidate problematic groups of components or nodes, and (iii) answer detailed questions about those components or nodes using dynamically deployed monitoring or analysis functions. VScope can operate on any set of nodes or software components and thus, can be applied within a tier, across multiple tiers, and across different software

levels.

2. *Guidance.* Replacing the current manual 'problem ticket' mechanisms used in industry, VScope-based troubleshooting is directed by on-line 'guidance', realized by the *Watch* and *Scope* operations that first detect abnormal behavior, followed by exploring candidate sources for such behavior, and only then lead to more detailed queries on select entities. The current implementations of *Watch* and *Scope* support 'horizontal guidance' to track potential problems across different tiers of a multi-tier application, and 'vertical guidance' to understand whether problems are caused by how applications are mapped to machines.

3. *Distributed Processing Graphs (DPGs).* All VScope operations are realized by DPGs, which are overlay networks capable of being dynamically deployed and reconfigured on any set of machines or processes, supporting various types of topologies and analysis functionalities [21, 38, 23]. First introduced in our previous work [187], where we proposed the basic architecture of DPGs and analytically analyzed their cost&benefits in data center management, this chapter presents their implementation, APIs, and commands, based on which we then build VScope's troubleshooting functionality.

VScope's capabilities and performance are evaluated on a testbed with over 1000 virtual machines (VMs). Experimental results show the VScope runtime to only negligibly perturb system and application performance, and requiring mere seconds to deploy 1000 node DPGs of varying topologies. This results in fast operation for on-line queries able to use a comprehensive set of both application- to system/platform-level metrics and a variety of representative analytics functions. When supporting algorithms with high computation complexity, VScope serves as a 'thin layer' that occupies no more than 5% of their total latency. Further, by using guidance that correlates system- and application-level metrics, VScope can locate problematic VMs

that cannot be found via solely application-level monitoring, and in one of the use cases explored in the chapter, it operates with levels of perturbation of over 400% less than what is seen for brute-force and most sampling-based approaches.

The remainder of this chapter is structured as follows. Section 4.3 introduces a typical real-time multi-tier application. Our goal and non-goals are described in Section 3.2.1. VScope design and implementation are in Section 3.2. We evaluate VScope in Section 3.3 and discuss the use cases in Section 5.4. We describes related work in Section 3.5 and concludes the chapter in Section 4.9.

3.2 System Design and Implementation

3.2.1 Goals and Non-Goals

The design of VScope is driven by several goals: (1) *flexibility*: to initiate, change, and stop monitoring and analysis on any set of nodes at any time, supported by operators for dynamically building and controlling user-defined actions for runtime troubleshooting; (2) *guided operation*: programmable methods for detecting potential problems and then tracking interactions that may contribute to them, between tiers and across software levels, thereby focusing troubleshooting to reduce overheads and improve effectiveness; and (3) *responsiveness and scalability*: to deploy troubleshooting methods with low delay at scales of 1000+ nodes.

VScope does not replace operator involvement, but aims to facilitate their troubleshooting efforts. Further, while VScope may be used to seek the root causes of failures, its current implementation lacks functionality like an off-line diagnostic database and a rich infrastructure for determining and using decision trees or similar diagnostic techniques [58]. Further, the methods presently implemented in VScope focus on persistent performance problems that will likely render an application inoperable after some time, i.e., when there are frequent or repeated violations of certain performance indicators that persist if they are not addressed. Having determined

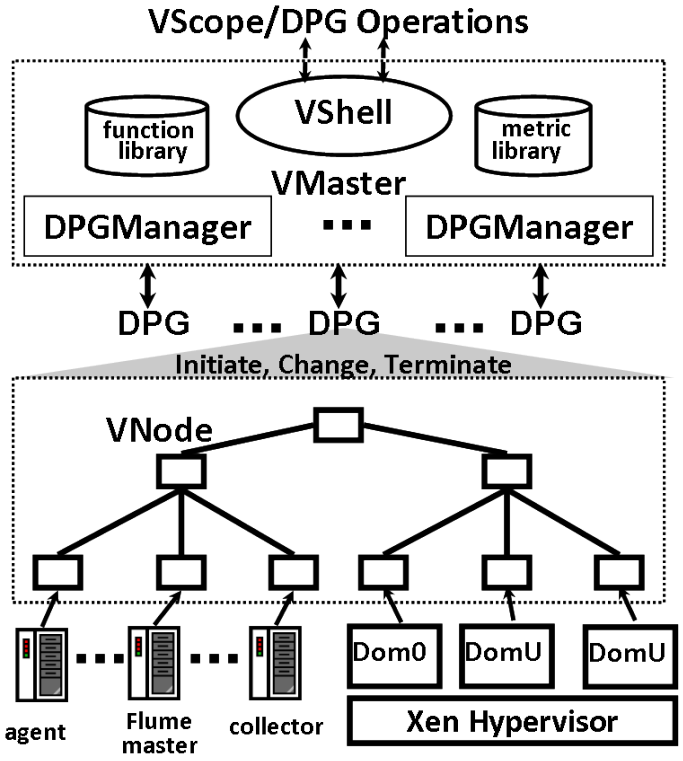

Figure 20: VScope System Overview

potential sources of such problems, VScope may trigger certain actions for mitigation or recovery, but it currently assumes such functionality to be supported by other subsystems (e.g., inherent to specific applications/tiers or software levels) or housed in some external system for problem resolution.

3.2.2 VScope Overview

The system architecture of VScope is depicted in Figure 20. The machines (VMs or physical machines) in the target application are managed by a server called *VMaster*. Operators use VScope operations, DPG commands, or scripts with the DPG API, in a console called *VShell* provided by *VMaster*. *VMaster* executes those commands by deploying DPGs on requested machines to process their monitoring metrics, and it returns results to operators. In detail, it starts a *DPGManager* to create a new DPG, which essentially, is an overlay network consisting of processing entities named

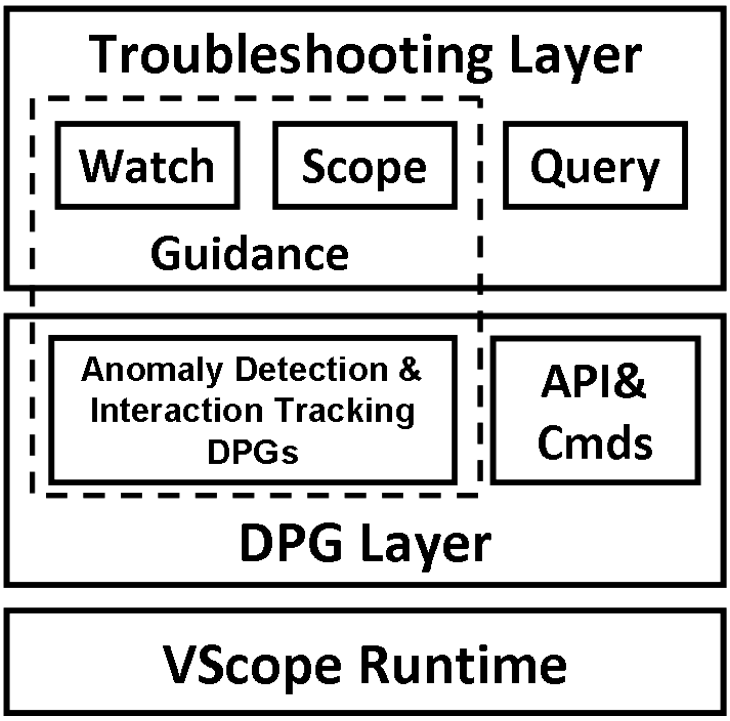

Figure 21: VScope Software Stack

VNodes residing on application machines. The *DPGManager* dynamically deploys *VNodes* equipped with assigned functions on specified machines, and connects them with a specified topology. *VNodes* collect and process monitoring metrics, transmit metrics or analysis results to other *VNodes* or the *DPGManager*, which in turn relays results to *VMaster*. *DPGManager* can initiate, change, or terminate its DPG on-the-fly.

In *VMaster*, the *metric library* defines monitoring metric types and associated collection functions. The *function library* defines the user-defined and default metric analysis functions, including those used in guidance (see Section 3.2.5). The above metrics and functions can be dynamically deployed into DPGs for various troubleshooting purposes.

The VScope software stack, described in Figure 21, has three layers. The troubleshooting layer exposes basic operations in *VShell*: *Watch*, *Scope*, and *Query*, which

Table 6: Arguments of Watch(*Optional)

Argument	Description
nodeList*	a list of nodes to monitor
metricList	a list of metric types
detectFunc*	detection function or code
duration*	duration
frequency*	frequency

will be described in Section 3.2.3. The *Watch* and *Scope* operations constitute the *guidance mechanism*, where *Watch* notifies the operator when and where end-to-end anomalies happen, and *Scope* provides the potential candidate nodes contributing to the anomalies. Operators can then use *Query* for in-depth analysis on those candidates yielded by guidance. These operations are built upon the DPG layer. In particular, the guidance mechanism (*Watch* and *Scope*) relies on an anomaly detection DPG and on interaction tracking DPGs. The DPG layer also exposes API and management commands to offer finer grain controls and customization. The lowest layer, the VScope runtime, is comprised of a set of daemon processes running on all nodes participating in the VScope system (i.e., the machines hosting the application and additional management machines running VScope). This runtime maintains the connections between machines and implements dynamic DPG creation and management. In virtualized data centers, the VScope runtime can be installed in hypervisors (e.g., Dom0 in Xen), in the virtual machines hosting the application(s) being monitored, in both, and/or in specialized management engines [116, 135]. Our testbed uses a VScope installation in the hypervisor and in the VMs hosting the Flume application.

3.2.3 Troubleshooting Operations

Watch.

The *Watch* operation monitors a list of metrics on a set of nodes,[2] and it applies to them an anomaly detection function in order to detect and report anomalous behaviors for any specified metrics. The parameters of the *Watch* operation described in Table 6 show its ability to monitor metrics on any VScope node, using some designated detection function specified with *detectFunc*. Sample functions used in our work include thresholding key performance indicators (KPI) such as request latency and statistics like those based on entropy described in [188]. The frequency and duration of the *Watch* operation are also configurable. In our Flume application, the *Watch* operation continuously executes on all of the Flume *agent* nodes, monitoring their end-to-end message latencies and detecting the nodes with latency outliers. Internally, *Watch* is implemented using an anomaly detection DPG explained in Section 3.2.5.

Scope.

The *Scope* operation (described in Table 7) discovers a set of nodes interacting with a particular node specified by argument *source*, at a specified time by argument *timestamp*. This operation guides troubleshooting by informing operators which nodes are related to the problematic node when the anomaly happens. Based on this guidance, operators can deploy DPG on those nodes (or subset of them) for further diagnosis using the *Query* operation. For instance, for the our Flume application, we have 'horizontal guidance' that identifies the HBase *region servers* a specified Flume *agent* is interacting with through a Flume *collector*, and 'vertical guidance' tracks the mappings between a physical machine and VMs hosted by it. By default, the output of *Scope* is a list of nodes directly interacting with the *source*. *distance* and *direction* are optional arguments. The former specifies indirect interactions by setting the value >

[2] A node is a physical or a VM running the VScope runtime in example application.

Table 7: Arguments of Scope(*Optional)

Argument	Description
nodeList*	a list of nodes to explore
graph	name of interaction graph
source	node in interest
timestamp*	interaction at a specific time
distance	number of edges
direction*	backward, forward or both

1, and the latter specifies the direction of interaction, for instance, 'receiving requests from' or 'sending requests to'.

In a nutshell, *Scope* works by searching an in-memory, global graph abstraction that describes interactions between every pair of nodes. Multiple types of interaction graphs are supported, covering a range of interactions from event level to network and system levels. These are shown in Table 12 and is specified by argument *graph*. The creation and continuous update of the global graph is implemented using an interaction tracking DPG explained in Section 3.2.5.

Query.

The *Query* function collects and analyzes metrics from a specified list of nodes, and provides results to query initiators. *Query* has two modes: *continuous* mode and *one-shot* mode, the latter being helpful when running monitoring or analysis actions that have high overheads. This is in contrast to the *Watch* operation which is always continuous because it is designed to monitor high level metrics to report on the target's global health. *Query* is flexible in that one can specify any group of nodes to analyze using *nodeList* and in addition, state the exact function to be used as *queryFunc* (see Table 8).

Table 8: Arguments of Query(*Optional)

Argument	Description
nodeList*	a list of nodes to query
metricList*	a list of metric types
queryFunc	analytics function or code
mode*	continuous or one-shot

3.2.4 Flexible DPGs

DPG as the Building Block.

All VScope operations described in Section 3.2.3 are implemented via DPGs. A DPG consists of a set of processing points (*Vnodes*) to collect and analyze monitoring data. It can be configured in multiple topologies to meet varying scale and analysis requirements. For example, it can be configured as a hierarchical tree or as a peer-to-peer overlay or, for smaller scales, as a centralized structure. Managed by a *DPGManager*, a DPG can be dynamically started on a specified set of nodes, where each *VNode* runs locally on a designated node and executes functions specified in VScope operations. These functions are stored as binaries in the *function library*, and they can be dynamically linked. As a result, DPGs are flexible in terms of topology, functions, and metric types. As illustrated in Figure 22, DPG outputs can be (i) presented immediately to the VScope user in *VShell*, (ii) written into rotating logs, or (iii) stored as off-line records in a database or key-value store. The last two configurations are particularly important when historical data is needed for troubleshooting. The use case in Section 5.4.2 uses rotating logs to store past metric measurements.

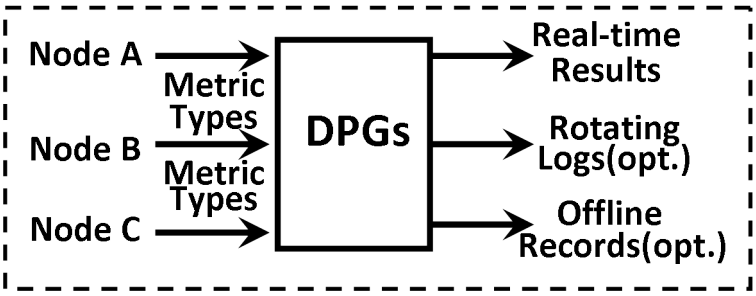

Figure 22: Black Box View of DPG

Table 9: Pseudo Functions for DPG API

DPG create (list, topology, spec)	Create a DPG with a specified topology
int add (src, dst, DPG)	Add a link from *VNode* src to *VNode* dst
int assign (func, spec, list, DPG)	Assign function to a list of *VNodes*
int start (DPG)	Start a DPG
int stop (DPG)	Stop an operating DPG
int insert (new, src, dst, DPG)	Insert a new *VNode* between existing *VNodes*
int delete (src, dst, DPG)	Delete a link from *VNode* src to *VNode* dst

DPG API and Management Commands.

Figure 23 describes the DPG core API and sample topologies, with details shown

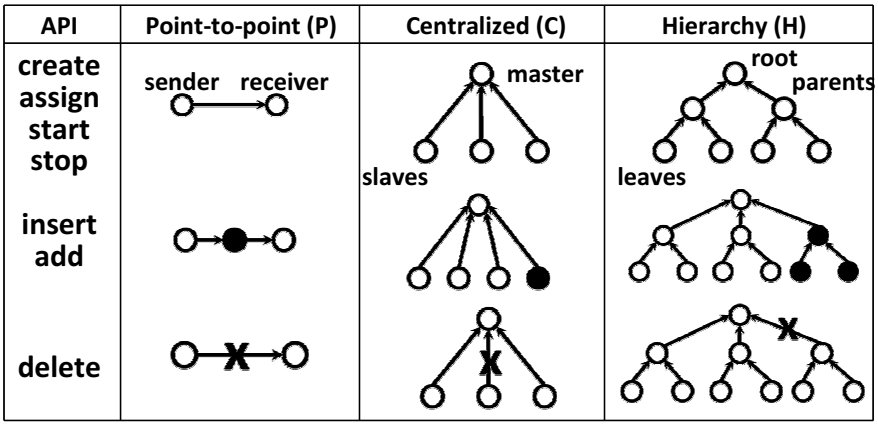

Figure 23: DPG API and Topologies

in Table 9. The *create()* method automatically creates any size topology of type point-to-point (P), centralized (C), or hierarchy (H) for some specified list of nodes. Topology specifics are configurable, e.g., besides the number of nodes, one can specify the branching factor of a hierarchical topology. *Create()* returns a unique DPG ID for reference in subsequent operations, and in the *assign()* method, the parameter *func* is a registered function ID. When a DPG is running, one can call the *assign()* to change the functionality on any *VNode* or use the *insert()* and *delete()* methods to change the DPG. The DPG API is exposed as commands in *VShell*, as well, and there are additional auxiliary management commands such as *list* (listing metric types, functions, or DPGs) and *collect* (returns the metric collection function).

Though operators can just use VScope operations without knowing the underlying DPG logic, new topologies, new operations and customization of existing functionality can be added easily through direct use of DPG APIs, which is not described in detail here because of space constraints.

3.2.5 Implementation

VScope Runtime.

The VScope runtime is implemented with EVPath [70], a C library for building active overlay networks. Metric collection uses standard C libraries, system calls, and JMX (at application level). Metrics are encoded in an efficient binary data format [70], and a standard format template is used to define new metric types. Built-in metrics and functions are listed in Table 10 and Table 21. As shown in the tables, VScope has a comprehensive set of metrics across application, system and platform levels, and a variety of representative analytics functions that are implemented with standard C libraries and other open source codes [65]. The DPGs associated with these functions have different topologies. For instance, *Pathmap*, PCA (Principle Component Analysis) and K-Clustering are implemented as centralized DPGs, as they require global data.

Table 10: Basic Metrics

Level	Basic Metrics
Application	E2E Latency, JMX/JVM Metrics Flume/HBase/HDFS INFO Logs
Virtual Machine	VCPU, Memory, I/O Metrics Network Traffic, Connections
Dom0 & System	CPU, I/O and Memory Metrics Paging, Context Switch Metrics

End-to-End Anomaly Detection.

The *Watch* operation is implemented using a DPG with a hierarchical topology in which the leaves are all of the nodes of the web log analysis application. This DPG

Table 11: Built-in Functions

	DPG	Algorithms
Watch	Hierarchy	MAX/MIN/AVE, Entropy, Top-K
Scope	Centralized	Pathmap[21]
Query	Centralized	K-Clustering, PCA

collects the end-to-end latency on each Flume *agent*, which is defined as the duration between the time when a new log entry is added and the time it (or its associated result) appears in HBase. This is measured by creating a test log entry on each *agent*, querying the entry in HBase and computing the difference. The latencies are then aggregated through the tree using Entropy-based Anomaly Testing (EbAT) [188], a lightweight anomaly detection algorithm, to output the agents that are outliers.

Interaction Tracking.

Table 12 shows built-in global graphs supported by *Scope*, covering a range of inter-

Table 12: VScope Interaction Graphs

	Interaction	DPG
Causality Graph	Event Flow between Nodes	Centralized Using Pathmap
Connection Graph	Network Connection	Distributed Using Netstat
Virtual Graph	Dom0-DomU Mapping	Distributed Using Libvirt
Tier Graph	Dependency between Tiers	Distributed Static Config.

actions from event level to network and system levels. For each graph type, in our implementation, a DPG is deployed and

continuously run on all the nodes to construct and update the corresponding graph structure in *VMaster*. There are two ways to track the global interactions, *centralized* or *distributed*. For interactions like the causality graph implemented using

Pathmap [21], a DPG collects metrics from leaves, compresses them at intermediate nodes, and then constructs the graph at the DPG root. An alternate distributed implementation of graph construction uses parallel analysis in which the leaves analyze metrics to generate a local graph (e.g., in the connection graph, it is the ingress and egress connections on a node), the local graphs are aggregated at parent nodes to create partial graphs which are finally aggregated at the root to produce the global graph. The current prototype uses adjacency lists to represent graphs and employs the parallel algorithm shown in Algorithm 1 to merge adjacency lists.

Algorithm 1: Parallel Graph Aggregation

1. On each leaf node, generate an adjacency list sorted by vertex IDs, and send it to the parent
2. On each parent or root node, merge n sorted adjacency lists as follows:
    1. Create an array P with size of n storing the first vertex ID in each adjacency list.
    2. If multiple IDs in P are the same and they are the smallest, merge their records into a new record, else take the record with the smallest vertex ID in P as the new record.
    3. Place the new record into the merged adjacency list.
    4. Update P to reflect the next record in each adjacency list.
    5. Repeat ii to iv until all the records in n adjacency lists are visited.

3.3 Experimental Evaluation

Experiments are conducted on a testbed consisting of 1200 Xen VMs hosted by 60 physical server blades running Ubuntu Linux (20 VMs per server). Every server has a 1TB SATA disk, 48GB Memory, and 16 CPUs (2.40GHz). Each VM has 2GB memory and at least 10G disk space.

Table 13: VScope Runtime Overheads

VNode Number	CPU Usage Increase	Memory Usage Increase
1	$< 0.01\%$	0.02%
5	$< 0.01\%$	0.02%
50	$< 0.01\%$	0.03%

3.3.1 VScope Base Overheads

To assess the basic costs of using VScope, we install it on a VM and vary the number of *VNodes* used in that VM. Each such *VNode* collects all of the metrics shown in Table 10, sending each metric to a separate DPG. As shown in Table 13, CPU and Memory overhead is negligible even when there are 50 *VNodes* (i.e., 50 concurrent DPGs) running. With continuous anomaly detection and via interaction tracking, VScope imposes only 0.4% overhead on the end-to-end latency of application described in Section 5.4. Note that in Section 5.4, VScope operations using tracing or logging analysis may incur considerable overhead, but this is due to the high costs of logging and tracking. The 'thin' VScope layer itself does not add notable costs, but instead, its use should reduce the inevitably high overheads of heavier weight, detailed logging and analysis by taking those actions only where and when they are needed.

3.3.2 DPG Deployment

Fast deployment of DPGs is critical for time-sensitive troubleshooting. We evaluate this by measuring the latency for deploying a hierarchical DPG on more than 1000 VMs, each of which has one *VNode*. The topology has a height of 2, and the total number of leaf VMs varies from 125 to 1000.

As shown in Figure 3.3.2, deployment time (presented as latency at Y-Axis) rises as the size of the DPG increases. However, latency remains within 5 seconds even at the scale of 1000 VMs. This would be considered sufficient for the current troubleshooting

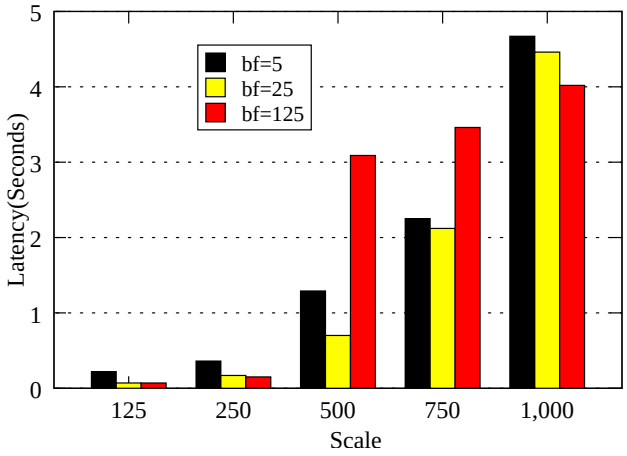

Figure 24: DPG deployment time w.r.t number of nodes

requirements (typically 1 hour) stated in [43], but it suggests the utility of future work on DPG reuse – to use and reconfigure an existing DPG, when possible, rather than creating a new one, or to pre-deploy DPGs where they might be needed. Deploying moderate scale DPGs with hundreds of nodes, however, usually happens within 1 second, suggesting that such optimizations are not needed at smaller scale. Also note that deployment latency varies with different branching factors (bf). At scales less than 750, deploying the DPG with bf 125 has larger latency than those with smaller bf values; this is because parent nodes construct their subtrees in parallel and the parents in the DPG with bf 125 have the biggest subtrees.

3.3.3 Interaction Tracking

The *Scope* operation relies on efficient methods for interaction tracking. We evaluate distributed DPG method (used for connection graph) by creating a two-level, hierarchical DPG with bf 25. We vary its number of leaves from 125 to 1000, and for this test, each VM has a randomly generated local interaction graph represented by an adjacency list with 1000 vertex entries with each vertex connected to 1000 randomly generated vertices. We measure the total latency from the time the first local graph is generated by leaf VMs to the time when the respective merged graph

is created at the root. We also measure the average time of local processing incurred during the per-node aggregation of connectivity graph information in order to study the dominant factor in total latency.

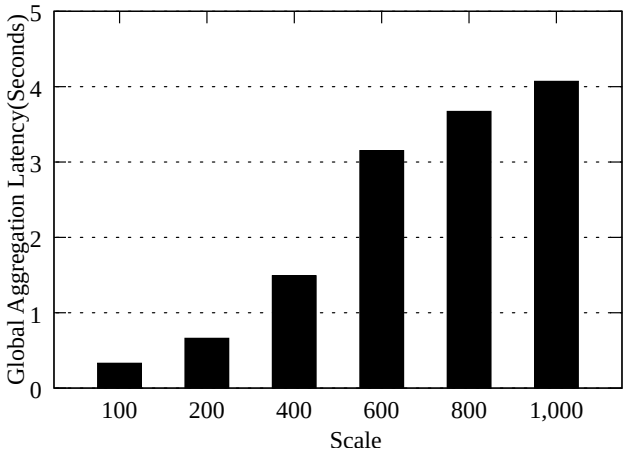

Figure 25: Explore operation latency w.r.t number of nodes

As shown in Figure 3.3.3, the total latency for generating a global graph increases as the system scales, but it remains within 4 seconds for 1000 VMs, where each VM has a 1000×1000 local connection graph. This means that the system can generate such a global graph at a resolution of every 4 seconds. Total latency is mainly due to the queuing and dequeuing time on *VNodes* plus network communication time. This is shown by the small measured local aggregation latency in Figure 3.3.3. At the same time, since these latencies increase linearly with the total number of inputs, parallel aggregation is a useful attribute to maintain for large scale systems. We also note that the local graphs occupy a fair amount of memory, which suggests opportunities for additional optimizations through use of more efficient internal data structures.

Finally, the analytics actions taken by *Scope* utilize the *Pathmap* for centralized interaction tracking. In Section 3.3.4, Figure 27 shows that it can generate a 1000 VM graph within 8 seconds.

In summary, the *Scope* operation's current implementation is efficient for the long running enterprise codes targeted in our work, but it may not meet the requirements

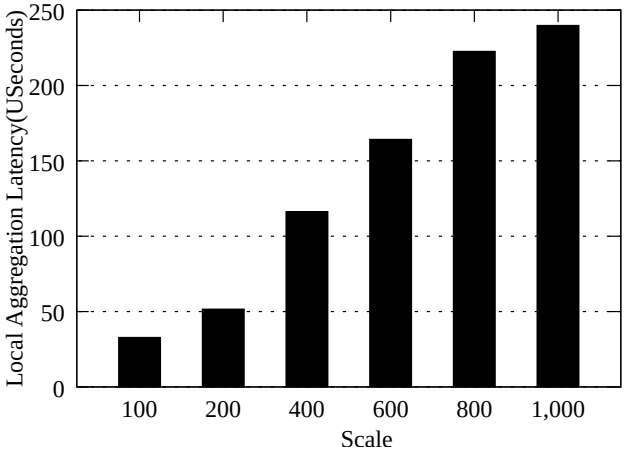

Figure 26: Global merge latency for guidance w.r.t number of nodes

of real-time codes such as those performing on-line sensing and actuation in highly interactive settings like immersive games.

3.3.4 Supporting Diverse Analytics

We use the algorithms in Table 21 as micro-benchmarks to measure the base performance of VScope operations. Tests randomly generate a 1000×1000 matrix of float numbers on each VM, and vary the size of the hierarchical DPG (bf=25) from 125 to 1000 leaf VMs. We measure the latency for analyzing the data on all leaf VMs at each scale. For centralized algorithms, the parent *VNodes* only relay the data. For the Top-K algorithm, we calculate the top 10 numbers. We conduct K-Means clustering with 5 passes.

Figure 27 shows latency breakdowns as well as the total latency of each function. In general, most of the algorithms operate within seconds, with increasing latencies for rising scales. Algorithms with high computational complexity are more costly, of course, but for such 'heavyweight' algorithms, especially for PCA, although the total latencies are over 1.5 minutes at the scale of 1000 VMs, VScope contributes only about 4.5% to these delays, and this contribution decreases as the system scales.

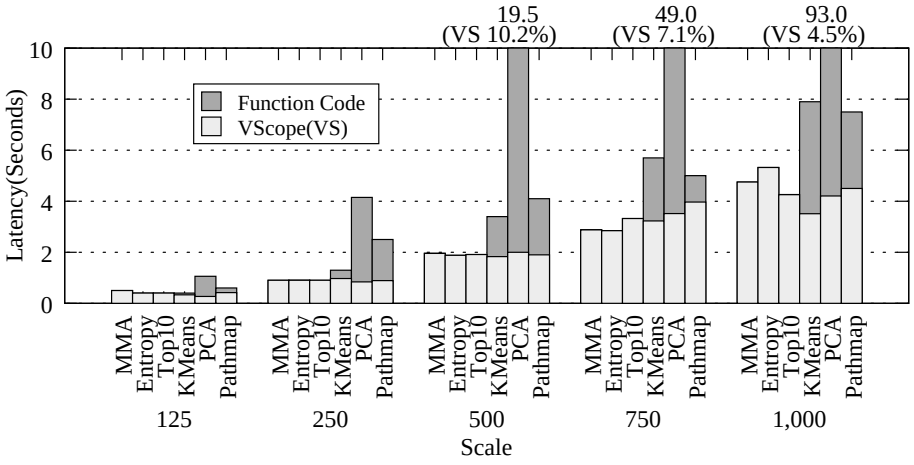

Figure 27: Analytics Microbenchmark Performance

3.4 Experiences with Using VScope

This section illustrates the utility of VScope for troubleshooting, using the application described in Figure 18. The application's Flume tier has 10 *collectors*, each of which is linked with 20 *agents*. The HBase tier has 20 *region servers*, and the HDFS tier has 40 *datanodes*.[3] Experiments use web request traces from the World Cup website [91] to build a log generator that replays the Apache access logs on each of 200 *agent* VMs. Each *agent* reads the new entries of the log and sends them to its *collector*. The *collector* combines the *ClientID* and *ObjectID* as the keyword, and the log content as the value, then places the record into HBase. The log generator generates 200 entries per second. The worst case end-to-end latency in the problem-free scenario is within 300 milliseconds.

The VScope runtime is installed on all of the VMs and in addition, on all physical machines (i.e., Xen's Dom0s). In accordance with standard practice for management infrastructures [116, 187], one additional dedicated VM serves as *VMaster*, and 5 dedicated VMs serve as parent *VNodes* in the two-level hierarchy DPGs used for troubleshooting. Two use cases presented below validate VScope's utility for efficient

[3]Each tier has one master node, and in HBase, 5 *region servers* serve as the ZooKeeper quorum. For simplicity, we do not 'count' masters when discussing scale.

troubleshooting.

3.4.1 Finding Culprit Region Servers

The first VScope use case crosses multiple tiers of the Flume application. The object is to find some 'culprit' *region server* exhibiting prolonged execution times. Those are difficult to detect with standard HBase instrumentation because debug-level logging in *region servers* to trace their request processing times [33] generates voluminous logs and high levels of perturbation for the running server(s). Troubleshooting using brute force methods might quickly find a culprit by turning on all of the *region servers*' debug-level logging and then analyzing these logs (in some central place), but this would severely perturb the running application. Alternative methods that successively sample some random set of servers until a culprit is found would reduce perturbation but would likely experience large delays in finding the culprit server. More generally, for multi-tier web applications, while bottleneck problems like the 'culprit' *region server* described above commonly occur, they are also hard to detect, for several reasons. (1) Dynamic connectivity – the connections between the Flume and HBase tiers can change, since the *region server* to which a collector connects is determined by the keyword region of the collector's current log entry. (2) Data-Driven concurrency – HBase splits the regions on overloaded *region servers*, causing additional dynamic behavior. (3) Redundancy – a *region server* is typically connected by multiple *collectors*. As a result, one *region server* exhibiting prolonged processing times, i.e., the 'culprit' *region server*, may affect the end-to-end latencies observed on many *agents*.

We synthetically induce server slowdown, by starting garbage collection(GC) in the Java Virtual Machine (JVM) on one of the *region servers*. This prolonged disturbance eventually slows down the Flume *agents* connected to the *region server* via their *collectors*. Experimental evaluations compare VScope, the brute-force, and

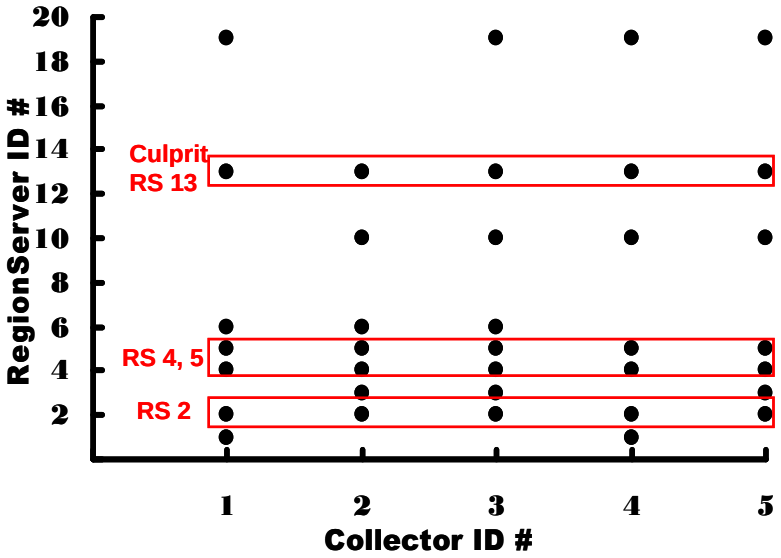

Figure 28: Find Culprit *Region Server*

the sampling-based approaches for finding the culprit *region server*. The VScope approach follows the 3 steps illustrated in Figure 29.

(1) A user at a *VShell* console issues a low-perturbation *Watch* operation to find which *agents* have prolonged end-to-end latencies. In Figure 28, the X axis shows the 5 problematic *collectors*, named 1 to 5, and the Y axis shows the 20 *region servers*, named RS 1 to RS 20. A dot on the graph indicates a connection, i.e., a *collector* has a network connection with a *region server* at that point in time. The red boxes indicate the common *region servers* connected to all of the *collectors*. RS 13 is the culprit.

(2) Use the *connection graph* (chosen from Table 12) and the *Scope* operation to find the connected *collectors* and the *region servers* to which they connect. These guidance actions involve using the *connection graph* as the *graph* parameter, using the problematic *agent* node as the *source*, and choosing 2 as the *distance* parameter. The output will be the *collector* and associated *region servers*. By iteratively 'Scoping' all anomalous *agents*, we find that they share 5 *collectors*. Furthermore, the *Scope* operation returns the set of *region servers* in use by these collectors and we can

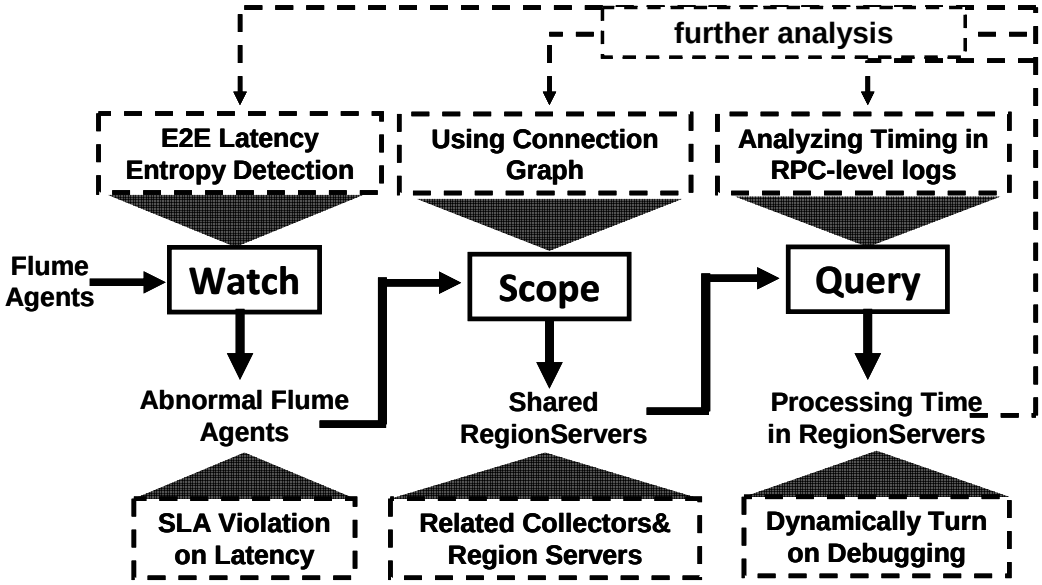

Figure 29: Steps using VScope operations

determine that they have 4 *region servers* in common. Therefore, we select those four as candidate culprits. Under the assumption of only one culprit *region server*, this operation will succeed because the culprit affects all of these *collectors*. While it will be rare to have multiple culprit *region servers* in a short period of time, in that case, more candidates may be chosen, but they still constitute only a small set of all *region servers*.

(3) Here, VScope has narrowed down the search for the problem *region server*, and we can use the *Query* operation to turn on debug-level logging on the candidates. We note that the *region servers* yielded by the *Scope* operation will always include the culprit, because VScope tracks all connections. The user will still have to carefully examine the region server logs to find the problem, but instead of having 20 candidates (the brute-force approach), there are just 4. If the examination is done sequentially (gather and examine logs one server at a time) to minimize perturbation, the user can expect on average to examine 2 logs (requiring 20 minutes of logging and .45GB

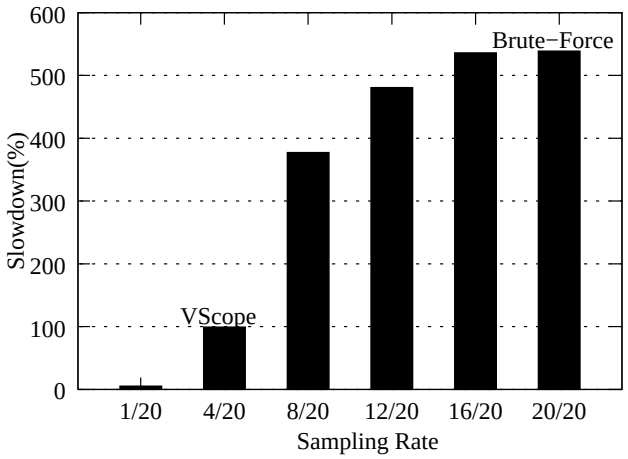

Figure 30: Slowdown w.r.t Sampling Rate

of data) with VScope, as opposed to 10 logs (requiring 100 minutes of logging and 2GB of data) with brute-force. If the log gathering is done in parallel to save time, the information provided by VScope allows just 4 logs (0.9GB) as opposed to 20 logs (4.1GB) to be retrieved by the brute-force approach. Note that, as shown in Figure 30, logging on multiple region servers simultaneously has a non-linear effect upon system performance. Simultaneous logging on only 4 servers (with VScope) slows the overall system down by 99.3%, but logging on all servers (brute-force) slows it by 538.9%.

Obviously, in-between approaches, i.e. random sampling, might log on more than one, but fewer than the total number of candidate region servers, hoping to trade off perturbation with "time-to-problem-discovery". In sum, the use of VScope has narrowed the set of possible bad region servers, thus improving the expected perturbation, log data sizes, and time to resolution in both the average and worst case.

These results validate the importance of VScope's guided operation that explicitly identifies the nodes on which troubleshooting should focus, in contrast to methods that use sampling without application knowledge or that employ non-scalable exhaustive solutions. They also demonstrate VScope's ability to assist with cross-tier troubleshooting. We note that, for sake of simplicity, this use case assumes the root

cause to be within the *region servers*. The assumption can be removed, and in that case, operators can apply further analysis as shown in Figure 29 by iteratively using VScope operations.

3.4.2 Finding a 'Naughty' VM

Previous research has shown the potential for running real-time application in virtualized settings [124]. However, VMs' resource contention on I/O devices can degrade the end-to-end performance of the application. A typical scenario is that some 'naughty' VM excessively uses a physical NIC shared by other VMs on the same physical host, thereby affecting the performance of the real-time VMs. Potential 'naughty' VMs could be MapReduce *reducers* exchanging voluminous data with a number of other nodes (e.g. *mappers*), or HDFS *datanodes* replicating large files. Contention could also stem from management operations like VM migration and patch maintenance [167].

There are remedies for contention issues like those above. They include migrating the 'naughty' VM and/or changing network scheduling. VM migration can involve long delays, and changes to VMs' network scheduling may involve kernel reboots not suitable for responsive management. The solution with which we experiment performs traffic shaping for the 'naughty' VM on-the-fly, in the

hypervisor, without involving guest VMs. To do so, however, support is needed to locate the troublesome VM. VScope running in the Dom0's of our virtualized infrastructure provides such support, Specifically, VScope deploys *VNodes* in the hypervisor's Dom0 and uses the virtualization graph shown in Table 12 to track mappings between VMs and hypervisors.

Experimental results with this use of VScope are obtained by emulating the 'naughty' VM issue by deploying a VM with a Hadoop *datanode* and *tasktracker*, on the host where a 'good' VM is running one of the 200 Flume *agents*. This scenario

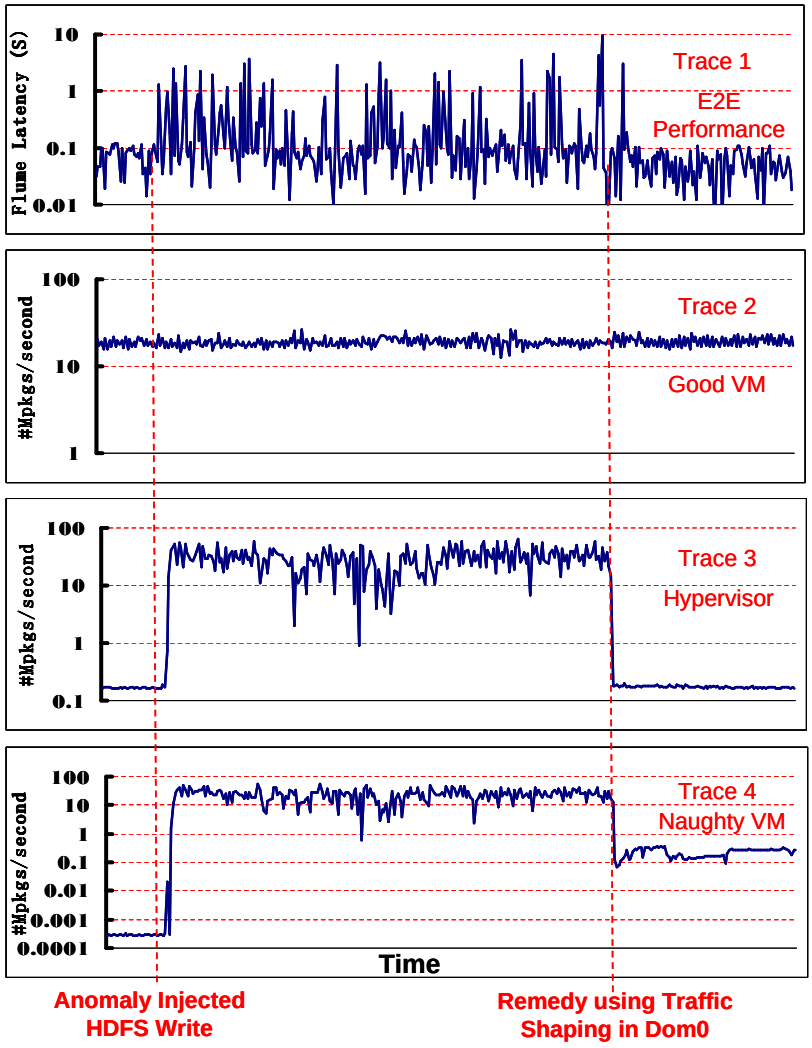

Figure 31: Using VScope to Find a 'Naughty' VM

is chosen to emulate co-running a real-time web log analysis with a batch system using Hadoop, with long term analysis based on the data generated by the real-time code. In this case, a problem is created by starting a HDFS benchmarking job called 'TestDFSIO write', which generates 120 2GB files with 4 replicas for each file in HDFS. This 'naughty VM' generates 3 files (we have 40 *slaves* in the Hadoop configuration. Every *slave* carries out 3 map tasks, each of which writes a 2G file to HDFS) and replicates them via the network. VScope is used to find that naughty VM, so that its communications can be regularized via Dom0 traffic shaping.

The monitoring traces in Figure 31 demonstrate VScope's troubleshooting process. Trace 1 presents the latency data generated by the *Watch* operation. Latency rises after the anomaly is injected. Using 1 second as the threshold for an end-to-end performance violation, after 20 violations are observed within 5 minutes, the *Watch* operation reports an anomaly and its location, i.e., the 'good' VM. After the anomaly is reported, troubleshooting starts for the VM by querying basic VM level metrics, including the *number of packages per second* represented by Trace 2,[4] where we find that metrics in the VM do not show abnormal behavior. In response, we use the *Scope* operation to find which physical machine is hosting the VM and then *Query* its aggregate packet rate. With these guided actions, we find in Trace 3 that the shared NIC is sending a large number of packets, in contradiction to the low packet rate in the 'good' VM. The next step is to further *Scope* the virtualization graph to find the other VMs running on the same physical host and then *Query* the network metrics of their VIFs.[5] The 'naughty' VM is easily found, because its respective VIF consumes the majority of the packets for the physical NIC, as shown in Figure 31 Trace 4. The correctness of the diagnosis obtained via VScope is demonstrated by applying traffic shaping in Dom0, which involves using TC to throttle the bandwidth of the 'naughty'

[4]We only show NIC-related metrics for succinctness.

[5]A VIF is the logical network interface in Dom0 accepting the packets for one VM and in our configuration, each VM has a unique VIF.

VM. It is apparent that this action causes the end-to-end latency of the good VM returns to normal in Trace 1. In Trace 3, the hypervisor packet rate goes down, and in Trace 4 the network consumption of the 'naughty' VM also sinks, as expected, but it still has its share of network bandwidth.

3.5 Related Work

Aggregation systems like SDIMS[193] and Moara[114] are most related to VScope in terms of flexibility. SDIMS provides a flexible API to control the propagation of reads and writes to accommodate different applications and their data attributes. Moara queries sub-groups of machines rather than the entire system. In both systems, flexibility is based on dynamic aggregation trees using DHTs (Distributed Hash Tables). VScope's approach is fundamentally different, in several aspects. First, VScope can control which nodes and what metrics to analyze; neither SDIMs nor Moara provides this level of granularity. SDIMS only controls the level of propagation along the tree, and Moara chooses groups based on attributes in the query (e.g., CPU utilizations). Second, the analysis functions in SDIMS and Moara are limited to aggregation functions, while arbitrary functions can be used with VScope, including those performing 'in transit' analysis. Third, like other monitoring or aggregation systems, including Ganglia[138], Astrolabe[179], and Nagios[78], SDIMS and Moara focus on monitoring the summary of system state, while VScope's goal is to provide in-depth troubleshooting solutions, including debugging and tracing, supported by basic metric aggregation like that performed in the *Watch* operation.

GWP[156], Dapper[165], Fay[73], Chopstix[42] are distributed tracing systems for large scale data centers. VScope is similar in that it can monitor and analyze in-depth system or application behaviors, but it differs as follows. First, instead of using statistical (Fay and Chopstix leverage *sketch*, a probabilistic data structure for metric collection) or random/aggressive sampling (as used in GWP and Dapper),

VScope can look at any set of nodes, making it possible to implement a wide range of tracing strategies (including sampling) through its guidance mechanism. Second, those tracing systems use off-line analysis, while VScope can analyze data on-line and in memory, to meet the latency restriction for troubleshooting real-time applications.

HiTune[64] and G^2[90] share similarity with VScope in that they are general systems for troubleshooting 'big-data' applications. HiTune extracts the data-flows of applications, using Chukwa for data collection and Hadoop for dataflow analysis. G^2 is a graph processing system that uses code instrumentation to extract runtime information as a graph and a distributed batch processing engine for processing the queries on the graph. VScope differs in its focus on on-line troubleshooting, whereas HiTune and G^2 are mainly for off-line problem diagnosis and profiling. Further, HiTune and G^2 are concerned with analyzing single applications, while VScope troubleshoots across multiple application tiers. Other troubleshooting algorithms and systems, such as Pinpoint[57], Sherlock[38], Project5[23], E2EProf[21], target traditional web applications. Plus, VScope is different because it is an infrastructure which can support various algorithms.

3.6 Conclusions

VScope is a flexible, agile monitoring and analysis system for troubleshooting real-time multi-tier applications. Its dynamically created DPG processing overlays combine the capture of monitoring metrics with their on-line processing, (i) for responsive, low overhead problem detection and tracking, and (ii) to guide heavier weight diagnosis entailing detailed querying of potential problem sources. With 'guidance' reducing the costs of diagnosis, VScope can operate efficiently at the scales of typical data center applications and at the speeds commensurate with those applications' timescales of problem development. The paper provides evidence of this fact with a real-time, multi-tier web log analysis application.

CHAPTER IV

EBAT & ANOMALY DETECTION ALGORITHMS

4.1 Introduction

The online detection of anomalous system behavior [55] caused by operator errors [150], hardware/software failures [154], resource over-/under-provisioning [117, 119], and similar causes is a vital element of operations in large-scale data centers and utility clouds like Amazon EC2 [2]. Given the ever-increasing scale coupled with the increasing complexity of software, applications, and workload patterns, anomaly detection methods must operate automatically at runtime and without the need for prior knowledge about normal or anomalous behaviors. Further, they should be sufficiently general to apply multiple levels of abstraction and subsystems and for the different metrics used in large-scale systems.

The detection methods [10, 8, 13] currently used in industry are often ad hoc or specific to certain applications, and they may require extensive tuning for sensitivity and/or to avoid high rates of false alarms. An issue with threshold-based methods, for instance, is that they detect anomalies after they occur instead of noticing their impending arrival. Further, potentially high false alarm rates can result from monitoring only individual rather than combinations of metrics. New methods [61, 23, 57, 40, 117, 21, 20, 38] developed in recent research can be unresponsive due to their use of complex statistical techniques and/or may suffer from a relative lack of scalability because they mine immense amounts of non-aggregated metric data. In addition, their analysis often require prior knowledge about application SLOs, service implementation, or request semantics.

This chapter proposes the EbAT – Entropy-based Anomaly Testing – approach to

anomaly detection. EbAT analyzes metric distributions rather than individual metric thresholds. We use entropy as a measurement to capture the degree of dispersal or concentration of such distributions. The resulting 'entropy distributions' aggregate raw metric data across the cloud stack to form 'entropy time series', and in addition, each hierarchy of a cloud (data center, container, rack, enclosure, node, socket, core) can generate higher level from lower level entropy time series. We then use online tools – spike detecting (visually or using time series analysis), signal processing or subspace method – to identify anomalies in entropy time series in general and at each level of the hierarchy. The current implementation employs wavelet analysis and visual spike detection. The hierarchical entropy time series analysis provides us with the ability to 'zoom in' to the components and metrics where anomalies may be occurring. A typical response to a detected anomaly is to trigger heavier weight diagnosis tools [21, 20, 61, 38, 57, 40, 23] that further identify anomaly causes and/or suggest suitable remedies.

EbAT constitutes a lightweight online approach to scalable monitoring, capable of raising alarms and zooming in on potential problem areas in clouds. Since the volume of monitoring data is exponentially reduced along the cloud hierarchy, the approach can easily scale as metric volume grows. When the metrics used are collected with black-box methods [23] like hypervisor-level monitoring, applications and systems can be monitored without client- or vendor-specific knowledge about their make, nature, or expected behavior. When applying detection to metrics collected with gray-box techniques [36], we exploit existing knowledge about certain hardware/software components or applications. In either case, there is no need for human involvement or intervention, thereby reducing operating costs, and there is minimum requirement of prior knowledge about hardware/software or of typical failure models. As a result, EbAT can detect anomalies that are not well understood (i.e., no prior models) or have not been experienced previously, and it can operate at any one and across multiple

of the levels of abstraction existing in modern systems, ranging from the hypervisor, to OS kernels, to middleware, and to applications.

In summary and compared to prior work on cloud anomaly detection, EbAT's technical contributions include the following:

1. a novel metric distribution-based method for anomaly detection using entropy;

2. a hierarchical aggregation of entropy time series via multiple analytical methods, to attain cloud scalability;

3. an evaluation with two typical utility cloud scenarios, to demonstrate the viability and accuracy of the proposed techniques, and to compare EbAT's methods with the threshold-based techniques currently in wide-spread use.

Expanding on 3) above, experimental results are attained with representative cloud applications that include RUBiS, a distributed set of auctioning services and a data intensive application written with Hadoop. The first case study compares the EbAT methods with a baseline method representing current best practices that use threshold-based anomaly detection. When comparing the precision, recall and F_1 accuracy score for anomaly detection attained with both techniques, results show that EbAT outperforms threshold-based methods with on average 18.9% in F_1 score and also does better by 50% on average in false alarm rate with a 'near-optimum' threshold-based method. The second case study uses a data intensive code to discuss interesting properties of EbAT and the manner in which its methods should be used.

The remainder of this chapter is organized as follows. Section 4.2 describes the research challenges. Section 4.3, Section 4.4, and Section 4.5 detail the EbAT approach. Experimental evaluation and discussion are described in Section 4.6 and 4.7. Section 4.9 present conclusions and future work.

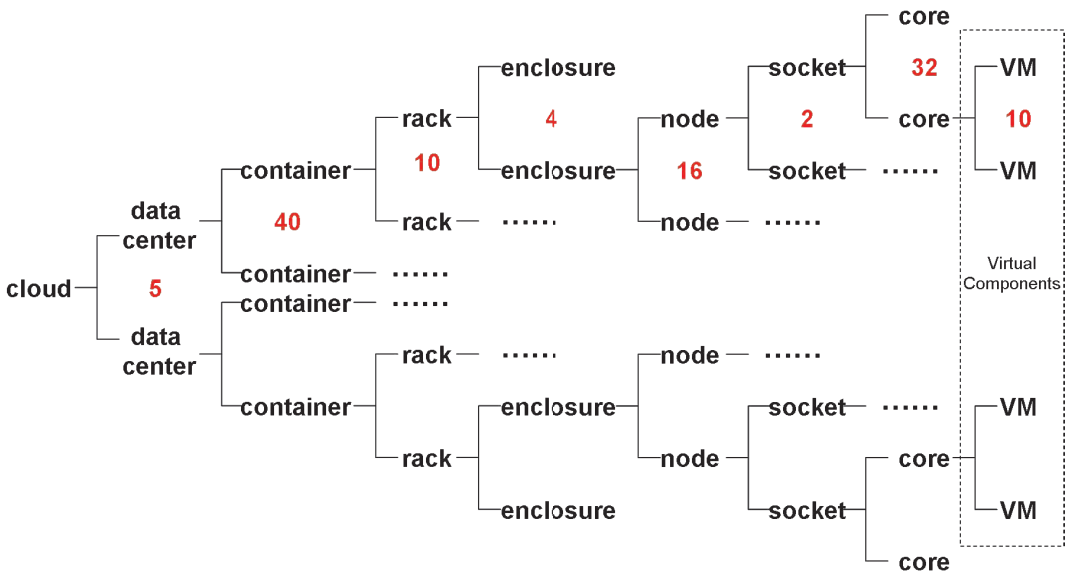

Figure 32: A cloud hierarchy

4.2 Problem Description

4.2.1 Utility Cloud Characteristics

A utility cloud's physical hierarchy is illustrated in Figure 32, where red numbers are typical quantities of components at each level. The hierarchical relationship between hardware components and virtual environment – data center, container, rack, enclosure, node, socket, core and virtual machines (VMs for short) – is typically configured statically at hardware levels. Considering virtual components, the simplified hierarchy used in our experiments describes VMs as direct children of nodes (hosts), but this can be generalized as shown in Figure 33:

1. *Exascale*: for up to 10M physical cores, there may be up to 10 virtual machines per node (or per core). Given a possibility of non-trivial number of sensors per node and additional sensors for each level of physical hierarchy, the total amount of metrics can reach exascale, 10^{18}.

2. *Dynamism*: utility clouds serve as a general computing facility. Heterogeneous

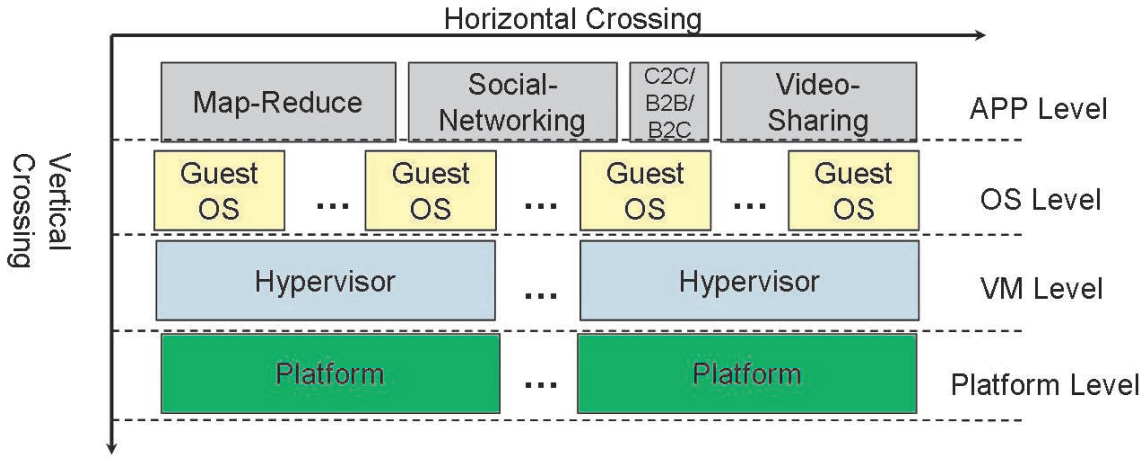

Figure 33: The functional view of a utility cloud

Table 14: Typical statistical approaches and their features

Technique	Scalability	Online	Black-Box	H-X	V-X
SLIC [61]	n	n	n	y	y
Nesting/Convolution [23]	n	n	y	y	n
Pinpoint [57]	n	n	n	y	n
Magpie [40]	n	n	y	y	n
Pranaali [117]	y	y	n	y	n
E2EProf[21]	n	n	y	y	n
SysProf [20]	n	n	y	y	n
Sherklock [38]	n	n	y	y	n
EbAT	**y**	**y**	**y**	**y**	**y**

applications include, but are not limited to, Map-Reduce, social networking, e-commerce solutions and multi-tier web applications, streaming applications and video sharing. While running simultaneously, these applications tend to have different workload/request patterns. Online management of virtual machines like live migration and power management make a utility cloud more dynamic than existing data center facilities. Further, utility clouds experience ontinuous churn of workloads with applications continuously entering and exiting the system.

4.2.2 Problem Definition

The scale and dynamic nature of future utility clouds require the solution of three major problems in online anomaly detection: metric aggregation, reliable detection, and anomaly zoom-in.

The *aggregation* problem involves the preprocessing and aggregating of raw metric data. A good solution to the aggregation problem should (1) reduce the data volume for further analysis so that it can scale well as the cloud grows. (2) It should retain valuable information for anomaly detection and identification. Finally, (3) aggregation should be 'horizontal crossing' and 'vertical crossing', meaning that metrics from different levels/components are collectively considered in order to detect anomalies.

The *detection* problem is the problem of designating, at runtime, the time instances at which the utility cloud is experiencing anomalies. An effective algorithm to the problem should have high detection and low false alarm rates. Further, it should be an unsupervised method given the lack of sufficient *a priori* knowledge about normal or anomalous behavior or conditions. Last, it should be an autonomic approach in order to contain personnel costs for increasing cloud scales.

The *zoom-in* problem consists of localizing anomalies so as to narrow down the search scopes for further diagnosing the causes of those anomalies. Zoom-in is important because it can speed up the problem finding process and can reduce the overheads of heavier weight, detailed diagnosis tools.

The effectiveness of an anomaly detection approach depends on the solutions to the above problems. As described in the next section, current methods do not already provide such solutions.

4.2.3 State of the Art

Threshold-Based Approaches

Threshold-based methods are pervasively leveraged in industry monitoring products [138, 78, 10, 8, 13]. They firstly set up upper/lower bounds for each metric. Those threshold values come from predefined performance knowledge or constraints (e.g., SLOs) or from predictions based on long-term historical data analysis. They can be set statically or dynamically. Whenever any of the metric observation violates a threshold limit, an alarm of anomaly is triggered.

Providing a moderate volume of metrics to operation teams with highly trained expertise, threshold-based methods are widely used with advantages of simplicity and ease of visual presentation. They however do not meet utility cloud requirements due to their following intrinsic shortcomings:

1. *Incremental False Alarm Rate (FAR)*: consider a threshold based method monitoring n metrics: m_1, m_2 ... m_n for each mi , if the FAR is r_i, the overall FAR of this method is $\sum_{i=1}^{n} r_i$. Thus, this means that when monitoring 50 metrics with FAR 1/250 each (1 false alarm every 250 samples), there will be $50/250 = 1/5$, i.e., 1 false alarm every 5 samples! Thus, the false alarm rate in threshold-based technique grows fast with increase in the number of monitoring metrics.

2. *Detection after the Fact*: consider 100 Web Application Servers (WAS) running the same service deficiently coded with *memoryleaks*. The memory utilization metrics from all those WASes may stay below their thresholds for a period of time as memory use is slowly increasing. Thus, no anomaly is detected. When one of the WASes raises an alarm because it crosses the threshold, it is likely that all other 99 WASes raise alarms soon thereafter, thereby causing a revenue disaster.

3. *Poor Scalability*: it is obvious that as we approach exascale volumes in monitoring metrics, it is no longer efficient to monitor metrics individually.

The lesson we learn from threshold-based approaches is that detecting anomalies by individual metric value threshold may not work well in exascale, highly dynamic cloud environments. Needed are new detection gauges and novel ways of aggregating metrics.

Statistical Methods

There exist many promising methods for anomaly detection, typically based on statistical techniques. However, few of them can deal with the scale of future cloud computing systems and/or the need for online detection, because they use statistical algorithms with high computing overheads and/or onerously mine immense amounts of raw metric data without first aggregating it. In addition, they often require prior knowledge about application SLOs, service implementations, request semantics, or they solve specific problems at specific levels of abstraction (i.e., metric levels). A summary of features of well-known statistical methods appears in Table 14 along with a comparison with EbAT.

Specifically, Cohen et al. [61] developed an approach in the SLIC project that statistically clusters metrics with respect to SLOs to create system signatures. Chen et al. [57] proposed Pinpoint using clustering/correlation analysis for problem determination. Magpie [40] is a request extraction and workload modeling tool. Aguilera. et al's [23] nesting/convolution algorithms are black-box methods to find causal paths between service components. Arpaci-Dusseau et al. [36] develop gray-box techniques that use information about the internal states maintained in certain operating system components, e.g., to control file layout on top of FFS-like file systems [148]. Concerning data center management, Agarwala et al. [21, 20] propose profiling tools, E2EProf and SysProf, that can capture monitoring information at different levels of

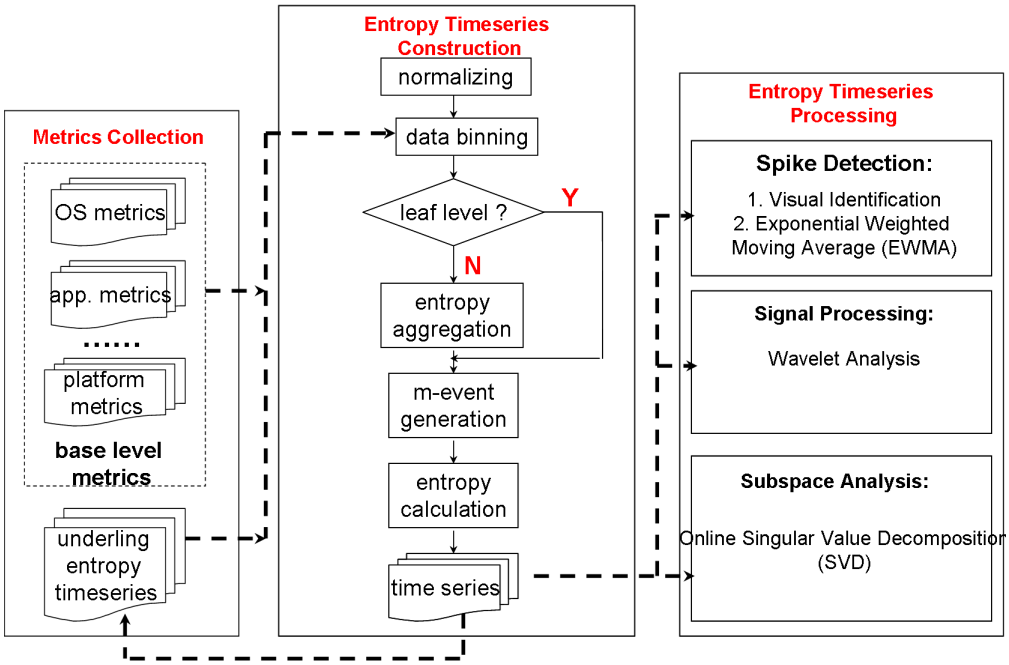

Figure 34: EbAT workflow

granularity. They however address different sets of VM behaviors, focusing on relationships among VMs rather than anomalies. Kumar et al. [117] proposed Pranaali, a state-space approach for SLA management of distributed software components. Bahl et al. [38] developed Sherklock using inference graph model to auto detect/localize internet services problems.

In contrast, our method (EbAT) aims to be address the scalability needs of Utility Clouds, providing an online lightweight technique that can operate in a black-box manner across multiple horizontal and vertical metrics. We leverage entropy-based analysis technique that has been used in the past for network monitoring [121, 122], but we adapt it for data center monitoring operating at and across different levels of abstraction, including applications, middleware, operating systems, virtual machines, and hardware.

4.3 EbAT Overview

The EbAT approach is depicted in Figure 34 as operating in three steps: *metric collection*, *entropy time series construction*, and *entropy time series processing*.

The *metric collection* happens at all components on each physical level of the hierarchy. The types of data collected depends on the level. A leaf component collects raw metric data from its local sensors. A non-leaf component collects not only its local metric data but also entropy time series data from its child nodes.

In the *entropy time series construction* step, data is normalized and binned into intervals. Leaf nodes only generate monitoring events (m-events for short) from its local metrics. Non-leaf nodes generate m-events from local metrics and child nodes' entropy time series, respectively. Those m-events are then counted at runtime to calculate the entropy time series of current components. Details about m-event and entropy calculation appear in Section 4.4.

In the *entropy time series processing* step, entropy time series are analyzed by one or multiple methods from spike detection, signal processing or subspace method in order to find anomalous patterns which are indications of anomalies in monitored system. Details are discussed in Section 4.5.

4.4 Entropy Time Series

4.4.1 Look-Back Window

As an online detection method, EbAT maintains a buffer of the last n samples' metrics observed. We call this buffer a *look-back window* which slides sample by sample during the monitoring process. The metrics in the look-back window at each time instance serve as inputs for pre-processing, m-event creation, and entropy calculation. The look-back window is used for multiple reasons. First, at exascale, it is impractical to maintain all history data. Second, shifts of work patterns may render old history data misleading or even useless. Third, the look-back window can be implemented in

high speed RAM, which can further increase detection performance.

4.4.2 Pre-Processing Raw Metrics

The monitoring data within each look-back window is first pre-processed and transformed into a form that can be readily used by EbAT. This pre-processing involves two steps explained below (also shown in Figure 34). Note that at each monitoring sample, multiple types of metric can be collected simultaneously , e.g. in our experiment, we collect CPU utilization, memory utilization and VBD read/write in each monitoring sample. In addition, the entropy values of child components can be collected simultaneously by non-leaf components. The pre-processing steps below are thus done for samples collected for every type of metric and/or child.

Step 1: Normalization In this step, EbAT transforms a sample value to a normalized value by dividing the sample value by the mean of all values of the same type in the current look-back window.

Step 2: Data binning Once normalization is complete, each of the normalized sample values are hashed to a bin of size m+1. This happens as follows. We predefine a value range [0,r], and split it into m equal-sized bins indexed from 0 to *m-1*. We define another bin indexed m which captures values larger than r. Both m and r are pre-determined statistically but are configurable parameters to the method. Each of the normalized values from Step 1 are put into a specific bin - if the value is greater than r, then it is placed in the bin with index m, else it is put in a bin with index, the floor of $sample_{value}$/(r/m). Thus, now each of the sample values are associated with a bin index number. This number is recorded and used to create m-events described in next subsection.

Figure 35 shows an example illustration of the above steps for a look-back window of size 3 with CPU and Memory utilization metrics.

Table 15: Definitions of B_{tj} and k

	B_{tj}	k
global	bin index of j^{th} child entropy	total# of children
local	bin index of j^{th} local metric	total# of local metrics

4.4.3 M-Event Creation

Once the sample data is pre-processed and transformed to a series of bin index numbers for every metric type and/or child, an m-event is generated that includes the transformed values from multiple metric types and/or child into a single vector for each sample instance. More specifically, an m-event of a component at sample t, E_t is formulated as the following vector description:

$$E_t =< B_{t1}, B_{t2}...., B_{tj}, ...B_{tk} > \quad (1)$$

where B_{tj} and k are defined as in Table 15,

The only restriction is that an entropy value and a local metric value should not be in the same vector of an m-event. The m-event for the example illustrated in previous subsection is shown in Figure 35.

According to the above definitions, at each component except for leaves, there will be two types of m-events:

1. *global m-events* aggregating entropies of its subtree, and

2. *local m-events* recording local metric transformation values, i.e. bin index numbers.

Within a specific component, the template of m-events is fixed, i.e. the types and the order of raw metrics in the m-event(local or global) vector are fixed. Therefore two m-events, E_a and E_b have the same vector value if they are created on the same component and $\forall$ j $\in$ [1, k],$B_{aj} = B_{bj}$

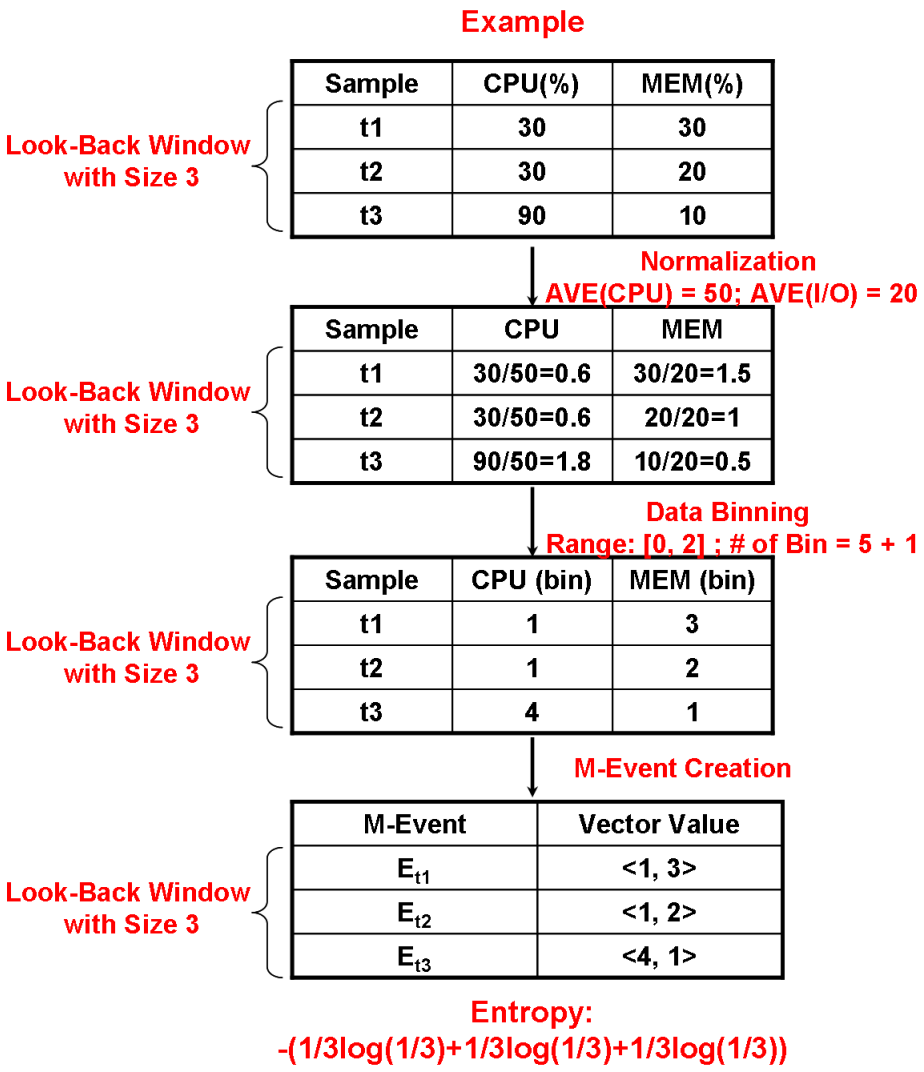

Figure 35: An illustration of the EbAT method

From above descriptions, an m-event is actually the representation of a monitoring sample. Therefore, on each component, there will be n m-events in the look-back window with size n. The next step is to extract the statistic characteristics of m-events in this look-back window.

4.4.4 Entropy Calculation and Aggregation

Entropy [163] is a widely used measurement that captures the degree of dispersal or concentration of random variable distributions. For a discrete random variable X with possible values $\{x_1, x_2 ..., x_n\}$, its entropy is:

$$H(X) = -\sum_{i=1}^{n} P(x_i) log P(x_i) \tag{2}$$

where $P(x_i)$ is the probability mass function of outcome x_i. $-logP(x_i)$ is called *surprisal* or *self-information* of x_i.

We deem the random variable as an observation E in the look-back window with size n samples. The outcomes of E are v m-event vector values $\{e_1, e_2 ..., e_v\}$ where v != n when there are m-events with the same value in the n samples. For each of these v values, we keep a count of the number of occurrence of that e_i in the n samples. This is designated as n_i and represents the number of m-events with vector value e_i. We then calculate $H(E)$ by the Formula 3. By calculating $H(E)$ for the look back window, we get the global and local entropy time series describing metric distributions for that look back window. Since, the look back window slides sample by sample as mentioned in Section IVA, $H(E)$ gets calculated at every monitoring sample for the corresponding look back window.

$$H(E) = -\sum_{i=1}^{v} \frac{n_i}{n} log \frac{n_i}{n} \tag{3}$$

In the case of local entropy calculation, the m-events represent the pre-processing results of the raw metrics. In the case of global entropy calculation, the m-events

represent the pre-processing results of the child entropy time series. While Equation 3 is one way to calculate global entropy using child entropy m-events, one can imagine other alternative aggregations too. We have in particular considered the following alternative formula (Equation 4) in our implementation which considers the combination of sum and product of the individual child entropies, and we show the evaluation comparison for the two approaches in Section 4.6.

$$A = \sum_{i=1}^{c} H_i + \prod_{i=1}^{c} H_i \tag{4}$$

Above, c is the number of children nodes, say, VMs on the same host (as in our experiment), and H_i represents the local entropy of the child i.

4.5 Entropy Time Series Processing

Entropy Time Series Analysis: After gathering the local/global entropy time series as described in the previous section, we can use data analysis methods to identify normal entropies and anomalous ones. The detection will reveal anomalous distribution changes in monitoring metrics. Appropriate techniques include 1) spike detection, 2) signal processing, and 3) subspace methods. The EbAT software framework permits one to choose any or multiple of those methods online, thereby enabling users to deal with individual methods' limitations and tune detection performance. The current implementation uses visual spike detection, as shown in Figure 37(a) for visually obvious spikes and wavelet analysis to identify abnormal patterns when visual detection is infeasible, as shown in Figure 37(b).

Zoom-In Identification is triggered when an anomaly is detected in the global entropy time series. By checking each child component's entropy in the global m-event, it is possible to reveal which one should be responsible for the anomaly. The process may be continued into lower levels until the anomalous metric data are found.

4.6 Evaluation with Distributed Online Service

Online services are an important class of applications in utility clouds. Using the widely used RUBiS benchmark deployed as a set of virtual machines, EbAT is evaluated for its' viability and in terms of effectiveness compared to the threshold-based methods in wide use. The goal is to detect synthetic anomalies injected into the RUBiS services. Effectiveness is evaluated using precision, recall and F_1 score as metrics which will be discussed in Section 4.6.4. Results show that EbAT outperforms threshold-based methods with on average 18.9% increase in F_1 score and we also do better by 50% on average in false alarm rate with the 'near-optimum' threshold-based method I.

4.6.1 Experiment Setup

RUBiS [53] is a distributed online service benchmark implementing the core functionality of an auction site. When used in the cloud context, its services are deployed in virtual machines mapped to the cloud's machine resources, as depicted in Figure 36.

The testbed uses 5 virtual machines (VM1 to VM5) on Xen platform hosted on two Dell PowerEdge 1950 servers (Host1 and Host2) with 2 x 2.8 GHz dual-core Xeon processors, 2 x 2.66GHz dual-core Xeon processors, Red Hat Enterprise Linux 5 OS, and 15GB RAM each. VM1, VM2, and VM3 are created on Host1. The frontend server processing or redirecting service requests runs in VM1. The application server handling the application logic runs in VM2. The database backend server is deployed on VM3. The deployment is typical in its use of multiple VMs and the consolidation of such VMs onto a smaller number of hosts.

A request load generator and an anomaly injector are running on two virtual machines, VM4 and VM5, on another server, Host2. The generator creates 10 hours' worth of service request load for Host1 where the auction site resides. The load emulates concurrent clients, sessions, and human activities. During the experiment,

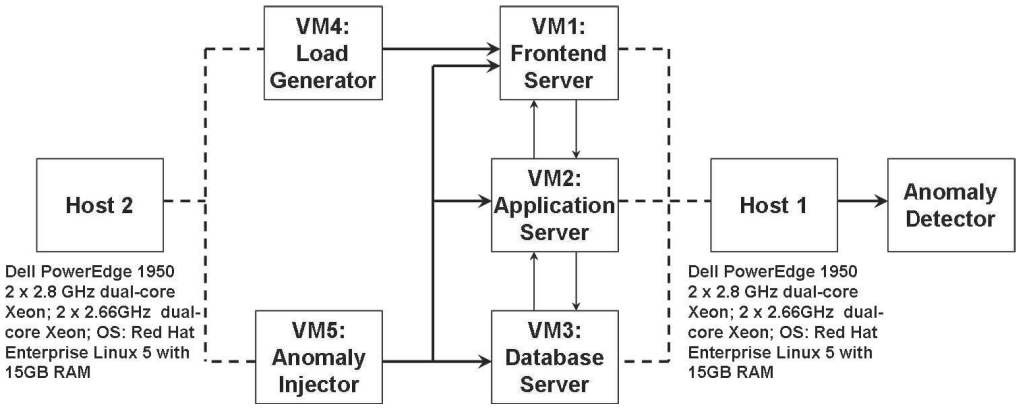

Figure 36: The experiment setup for RUBiS

Table 16: Anomalies Injected

Anomaly	Typical Source Type	# of Anomalies
Frontend Unaccessible	Operation Error	10
DB Server Unaccessible	Operation Error	16
APP Server Unaccessible	Operation Error	10
Frontend CPU Shortage	Resource Provision	3
Database CPU Shortage	Resource Provision	8
APP Server CPU Shortage	Resource Provision	3

the anomaly injector injects 50 anomalies into the RUBiS online service in Host1. Those 50 anomalies, as shown in Table 16, come from major sources of failures or performance issues in online services [150, 117, 119, 154]. We inject them into the testbed using a uniform distribution. The virtual machine metrics and the host metrics are collected using Xentop and analyzed in an anomaly detector. Anomaly detection applies threshold-based and EbAT's methods to online observations of the CPU utilizations of virtual machines and hosts (i.e., using black-box monitoring).

4.6.2 Baseline Methods – Threshold-Based Detection

We choose a threshold-based method as the baseline. This baseline method compares observed CPU utilization with a lower bound and higher bound threshold. Whenever the utilization goes below the lower bound threshold, or above the higher bound threshold, a violation to the thresholds is detected, and the baseline method will raise an alarm. Consecutive violations are aggregated into a single alarm. For our

evaluations, we consider two separate values of the thresholds. The first one is a near-optimum threshold value set by an 'oracle'-based method, and the other is a statically set threshold value that is not optimum but representative of how current state of the art real-world deployments use.

To obtain the near optimum threshold value, we feed the complete historical knowledge of the host CPU utilization to the baseline method. This historical trace matches exactly what the host will experience during the online RUBiS experimentation when the baseline method will be operating. From the historical CPU utilization trace, we calculate the histogram and from that, the lowest and highest 1% of the values are identified as representing outliers outside an acceptable operating range. The corresponding 1%boundary values are then chosen as the lower and upper bound thresholds. This is illustrated in Figure 38(b) for Host1. Specifically, 1.7% is chosen as the lower bound and 20.3% as the upper bound. 2% is chosen as the total value for outliers because the time taken by anomalies injected occupies 2% of the total experiment period. 2% is then appropriate in the sense that it can catch as many anomalies as possible without sacrificing accuracy, making the chosen threshold values near-optimum/ideal.

Figure 38(a) shows the results when applying these near-optimum thresholds to the host CPU utilization data for detecting anomalies. Red circles are violations when applying the baseline threshold method Consecutive red circles are deemed as a single anomaly because it may hold for some period of time. A successful alarm and the according actual anomaly injection may not happen at the same time because there is a latency between anomaly injection and its effects on the metrics being collected and analyzed.

For the non-optimum statically set threshold values, we choose 90% and 5% as the upper and lower bounds for the thresholds based on representative values used in state of the art deployments. We employ a similar exercise on the Host1 CPU

Table 17: Parameters for entropy calculation and aggregation

	look-back window size	range	# of bins
local entropy calculation	20	5	6
entropy I aggregation	100	5	5

utilization data using these threshold values to detect anomalies. The overall results are summarized in Table 18 which would be explained in Section 4.6.4.

4.6.3 EbAT Method Implementation

We apply the EbAT method implementation at the VM level and host level. As mentioned in Section 4.2, we deem VM1 to VM3 as the direct children of Host1. We thereby first calculate the local entropy time series in each VM and then aggregate them to global entropy time series for Host1. Two aggregations for global entropy described in Section 4.4.4 are evaluated and they are named "Entropy I" and "Entropy II", respectively (corresponding to Equations 3 & 4). The parameters chosen for entropy calculation and aggregation are presented in Table 17.

4.6.4 Evaluation Results

We use four measures [159, 16, 5], in statistics to evaluate the effectiveness of anomaly detection:

$$Precision = \frac{\#\ of\ successful\ detections}{\#\ of\ total\ alarms} \tag{5}$$

$$Recall = \frac{\#\ of\ successful\ detections}{\#\ of\ total\ anomalies} \tag{6}$$

$$Accuracy(F_1) = \frac{2 * precision * recall}{precision + recall} \tag{7}$$

$$False\ Alarm\ Rate\ (FAR) = \frac{\#\ of\ false\ alarms}{\#\ of\ total\ alarms} = 1 - Precision \tag{8}$$

Table 18: Experiment Results

Methods	# of Alarms	# of Hits	Recall	Precision	Accuracy	FAR
Entropy I	45	43	0.86	0.96	0.91	0.04
Entropy II	56	45	0.90	0.80	0.85	0.20
Threshold I	49	37	0.74	0.76	0.75	0.24
Threshold II	30	29	0.58	0.97	0.73	0.03

Precision is used to measure the exactness of the detection. Recall measures the completeness. Neither precision nor recall alone can judge the goodness of an anomaly detection method. Therefore we further use Accuracy, or F_1 score which is the harmonic mean of precision and recall, to compare the performances of EbAT and threshold-based methods. FAR is really 1-$Precision$.

A summary of results appears in Table 23, and detailed traces of entropy, raw metrics, anomalies injected and alarms are depicted in Figure 37 and 38(a). Overall, compared with the threshold-based detection methods, the use of EbAT results in on average 18.9% improvement in F_1 score. We also do better by 50% on average in false alarm rate with the 'near-optimum' threshold-based method I.

EbAT methods outperform threshold-based methods in accuracy and almost all precision and recall measurements. The only exception is that of precision with Threshold II, however Threshold II only detects 29 anomalies out of total 50, thus missing detection of many anomalies reflected through poor recall. Entropy II has worse FAR competed to Entropy I because the sum/multiply of entropies accumulate false alarms in each VM's entropy time series. The comparison between Entropy I and Threshold I, for example, indicates that EbAT's metric distribution-based detection aggregating metrics across multiple vertical levels has advantages over solely looking at host level, even when compared to an oracle detector.

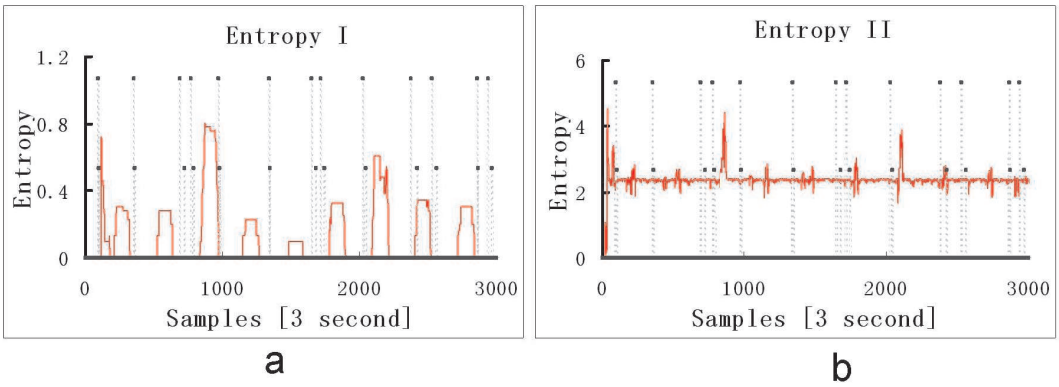

Figure 37: Fragment of Entropy I (a) and Entropy II (b) traces

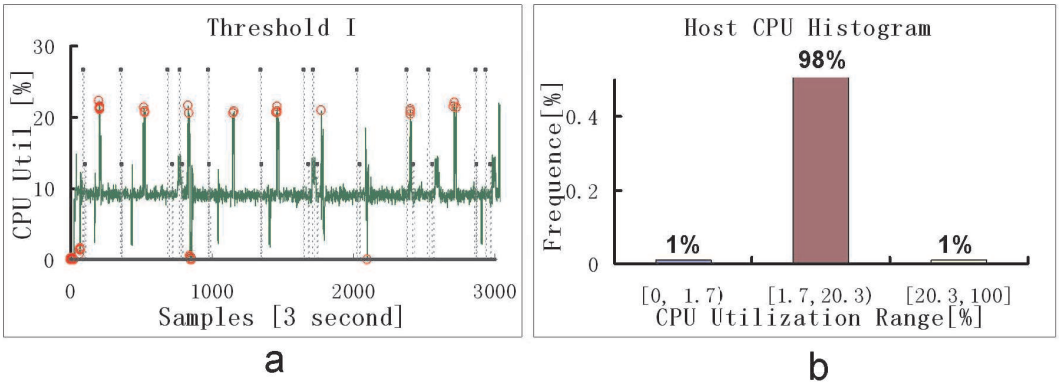

Figure 38: (a) A Fragment of Threshold I Trace, and (b) CPU Util. Histogram

4.7 Discussion: Using Hadoop Applications

Hadoop [4] is an implementation of the MapReduce framework [67] supporting distributed computing on large data sets. This section uses Hadoop to explore and illustrate select issues in using EbAT to monitor complex, large-scale cloud applications.

We deploy a Hadoop platform on three virtual machines named *master*, *slave*1 and *slave*2. Those VMs are hosted in one physical server with the same hardware configurations as in the RUBiS experiment. Master acts as the master node in MapReduce and also operates as a worker (i.e., running a worker process). Slave1 and slave2 are worker nodes. We run a distributed word counter application on the platform for 2 hours with 6 anomalies caused by application level task failures. EbAT observes CPU utilization and the number of VBD-writes and VBD-reads (virtual block device writes and reads), and calculates their entropy time series.

We first calculate entropies based on the CPU utilizations of the three VMs and aggregate those entropies as the host entropy. The parameters, look-back window size, range and bin number are presented in Table 17. Figure 39 shows each VM's CPU utilization trace. CPU utilizations are high when the associated VMs have tasks to run, but approach 0 in idle state. As a result, the entropy of the host, which is the aggregation of the three VM's entropies, exhibits high variability due to the high variations in raw metric values as seen in Figure 42(b). This demonstrates that it is not suitable to apply EbAT to metrics whose 'normal' behavior is highly variable. Such variations are simply transferred to the entropy trace, thereby affecting detection accuracy.

A more appropriate way of using EbAT with Hadoop applications is one in which its methods are applied to a derived metric. Specifically and as shown in Figure 40, although the VBD-write and VBD-read metrics keep increasing (will affect EbAT anomaly detection as mentioned above), they are highly correlated, as intuitively

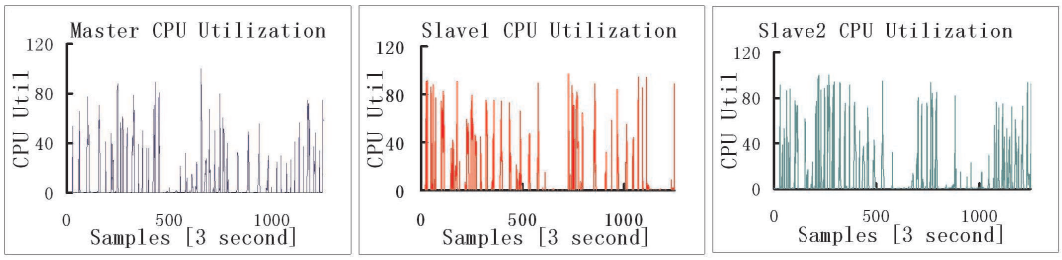

Figure 39: CPU utilizations of master, slave1 and slave2

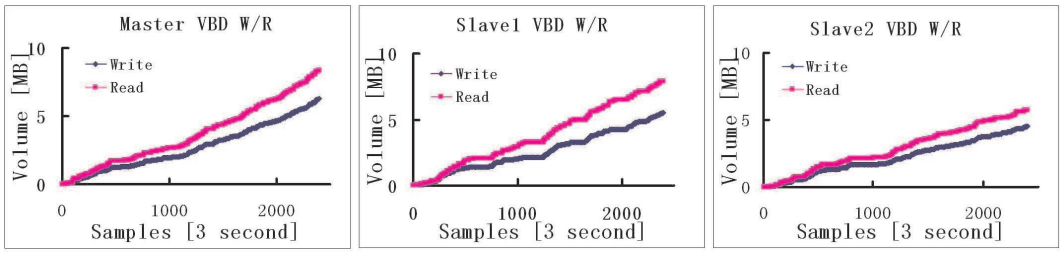

Figure 40: VBD-write and VBD-read in master, slave1 and slave2

obvious from the fact that data intensive codes implemented with Hadoop attain high performance by best using the aggregate disk bandwidth of the underlying machines on which they run. Thus, when calculating their correlation value traces as depicted in Figure 41, these correlations are stable most of the time, but have sharp decreases when anomalies occur. By applying EbAT calculations on correlation traces instead of raw metric traces, with the same parameters as those used for CPU utilization, we then attain the detection results presented in Figure 42(a). Here, six spikes in the whole host entropy time series reflect the six anomalies during the experiment period.

The above being a preliminarily empirical study on Hadoop use case, we are continuing to explore more sophisticated clustering techniques to find correlated metrics

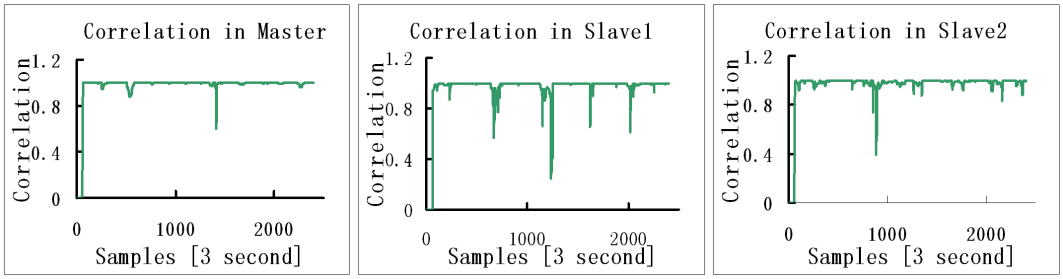

Figure 41: Correlations of VBD-write and VBD-read in master, slave1 and slave2

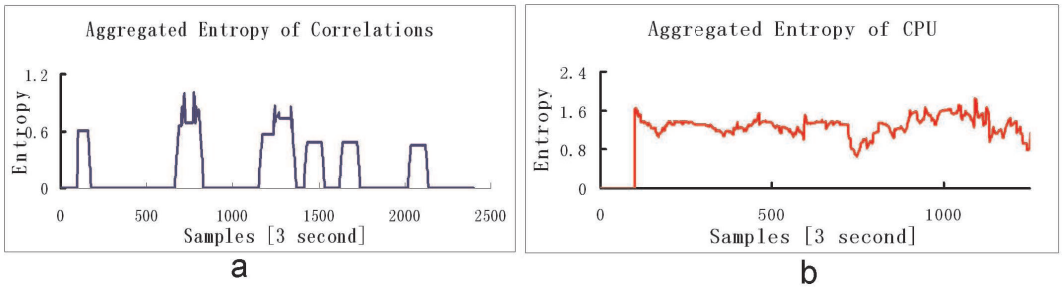

Figure 42: (a) Aggregated correlation entropies, and (b) CPU utilization entropy

and then track multiple entropies using EbAT on those correlation values.

4.8 Related Work

Research efforts on anomaly detection in utility cloud management can be categorized into two streams: data center management methods and performance debugging methods.

Concerning data center management, Agarwala et al. [21, 20] propose profiling tools, E2EProf and SysProf, that can capture monitoring information at different levels of granularity. They address different sets of VM behaviors, focusing on relationships among VMs rather than anomalies. Kumar, et al. [117] proposed Pranaali, a state-space approach for SLA management of distributed software components. Bahl, et al. [38] developed Sherklock using inference graph model to auto detect/localize internet services problems.

In performance debugging area, Cohen, et al. [61] developed an approach in the SLIC project that statistically clusters metrics with respect to SLOs to create system signatures. Chen, et al. [57] proposed Pinpoint using clustering/correlation analysis for problem determination. Magpie [40] is a request extraction and workload modeling tool. Aguilera. et al's [23] nesting/convolution algorithms are black-box methods to find causal paths between service components which is a different research purpose to ours. Arpaci-Dusseau et al. [36] develop gray-box techniques that use information about the internal states maintained in certain operating system components, e.g.,

to control file layout on top of FFS-like file systems [148]. Such methods can be employed, for instance, when entropy-based monitoring first detects potential reliability issues, but they are lacking in scalability and require prior knowledge about the applications or systems being monitored. A detailed comparisons appears in Table 14.

Entropy-based analysis has been used in network monitoring [121, 122], but must be adapted to the fact that data center monitoring should operate at and across different levels of abstraction, including applications, middleware, operating systems, virtual machines, and hardware, as evident from our work on zoom-in methods.

Most of current industry monitoring tools use fixed thresholds for anomaly detection. Fixed upper and lower bounds are determined apriori and remain constant during the entire process of anomaly detection. MASF [48] is one of the popular threshold-based techniques being adopted in industry. MASF applies thresholds to data segmented by hour of day, and day of week. As discussed earlier, these techniques have limitations of accuracy and false alarm rates due to their assumed data distributions, and limited adaptibility to changing workloads. They have poor scalability and lack of correlation analysis.

Prior academic work has developed many useful methods for anomaly detection, typically based on statistical techniques [61, 23, 57, 40, 117, 21, 20, 38]. A summary is provided by [55]. However, few of them can operate at the scale of future data center or cloud computing systems and/or have the 'lightweight' characteristic desired for online operation. Reasons include their use of statistical algorithms with high computing overheads and their use of onerously large amounts of raw metric information. In addition, they may require prior knowledge about application SLOs, service implementations, request semantics, or they are focused on solving certain well-defined problems at specific levels of abstraction (i.e., metric levels) in data center systems.

In contrast, the techniques we propose are lightweight techniques that systematically improve on the state of art techniques widely adopted in industry. We have proposed techniques to improve on current point fixed-threshold approaches, and we have also developed new windowing-based approaches that observe changes in the distribution of data. Our techniques are adaptable and can learn the workload characteristics over time, improving accuracy and reducing false alarms. They also meet the scalability needs of future data centers, are applicable to multiple contexts of data (catching contextual anomalies), and can be applied to multiple metrics in data centers.

4.9 Conclusions

EbAT is an automated online detection framework for anomaly identification and tracking in data center systems. It does not require human intervention or use predefined anomaly models/rules. To deal with the complexity and scale of monitoring, EbAT uses efficient m-events to aggregate different levels of metrics in clouds, leverages entropy-based metric distributions, time series diagnosis methods to detect anomalies at runtime, and zoom in detection to focus on possible areas of causes.

Future work concerning EbAT includes (1) extending and evaluating the methods for cross-stack (multiple) metrics, (2) evaluating scalability with large volumes of data and numbers of machines, and (3) continued evaluation with representative cloud workloads such as Hadoop.

Anomaly Detection is an important component for closed loop management in data centers. In this paper, we presented statistical approaches for online detection of anomalies. Specifically, we have presented methods based on Tukey and Relative Entropy statistics, and experimentally evaluated them. The proposed approaches are lightweight and improve upon prevalent Gaussian-based approaches.

CHAPTER V

VFOCUS: GRAPH-BASED GUIDANCE APPROACH

5.1 Introduction

Troubleshooting large scale distributed systems/applications in data centers is critical and challenging. As illustrated in Figure 32 and Figure 18, there can be millions of hardware entities (e.g. cores) or thousands of software components (e.g. region-servers). If there is no effective 'zoom-in' approach which can reduce the search space, troubleshooting could be like finding a needle in a haystack.

In Chapter II, The Monalytics research has proved the importance of 'zoom-in' analysis in reducing the troubleshooting time, i.e. TTI (Time To Insight), and monitoring data size. In Chapter III, The VScope research presents a series of ad hoc solutions to track interactions among components, and validates their effectiveness in terms of horizontal guidance and vertical guidance. Previous works [23, 21, 38] use dependency graph to track relationships among hardware/software components. However, the following challenges still remain

- `A framework for general troubleshooting guidance`: Previously proposed solutions are only effective for specific usage scenarios and target applications. For instance, VScope provides a set of built-in DPGs to track interactions. Sherlock and Orion focus on network services dependencies.Given the diversity of possible root causes and complex performance problem symptoms, those previous approaches fail to achieve broader applicability and better scalability. Therefore, a unified guidance framework is needed for effective zoom-in analysis.

- `A mechanism for analyzing interactions`: Previous works focus on capturing, not analyzing, interactions. Part of the reason is that they aim at detecting performance bottleneck, where 'knowing' the dependencies is often sufficient. However, as the interactions become more complicated and various troubleshooting scenarios other than bottleneck detection arise, the analysis of the interactions becomes necessary.

- `Tracking and analyzing interactions on-line`: Most of the previous works operate offline, and do not support responsive and on-line interaction tracking.

To address above issues, we propose VFocus , a framework for online interaction tracking and analysis.VFocus implements a generic guidance framework which is based on graph analysis, and enhances VScope's rather ad hoc guidance component described earlier.. Users can write various interaction tracking mechanisms by following the unified definitions of graph and snapshot. Users can also use the generic guidance operations to 'zoom-in' the search space of analysis. The guidance operations are realized by the graph analysis functions provided by VFocus. From VScope's perspective, VFocus extends its *Scope* function.

In VFocus research, we make following technical contributions:

- `A generic graph-based guidance framework`: we design and implement VFocus, an interaction graph construction and analysis framework, by which the users can generate graphs describing specific interactions they want to track.

- `Guidance operations using graph analysis algorithms`: VFocus provides basic guidance operations, *sort*, *group* and *explore*, to reduce the search space for troubleshooting. They are based on graph analysis algorithms, and can be efficiently conducted online in real-time.

- `Validate VFocus on real world data center traces and use cases`: we validate VFocus' guidance functions in three usecases. The first two usecases are based on real-world data center traces. We replay trace in our testbed and measure VFocus' performance on troubleshooting and overheads of operation. In the third use case we deploy VFocus on a running big-data application and evaluate its performance on troubleshooting the representative HBase data conflict issue. The experimental results from the three usecases show that VFocus can troubleshoot performance issues, such as VM migration failures with accuracy as high as 83%. The overhead of VFocus is low and the interference to the application is lower than traditional brute-force approach.

5.2 Graph-Based Troubleshooting

5.2.1 Systems as Graphs

Most services and applications in data centers are distributed systems, for instance, key-value stores, MapReduce platform and distributed file systems. In a distributed system, software/hardware components interact to realize its functionalities, which makes graph an intuitive means to understand the system's behavior. The components can be modeled as vertices while the interactions are modeled as edges, and both vertices and edges can have attributes. Graph modeling focuses on interactions, e.g. how the components of the system interact to affect the overall performance. The key challenge in troubleshooting data center systems is to understand the complex and dynamic interactions in large scale systems, and graph is a natural choice to address this challenge.

In troubleshooting context, there are many scenarios which are suitable for graph modeling, and they generally have two features: (1) to monitor multiple or potentially many entities and (2) to monitor multiple or potentially many interactions among entities to diagnose the performance issues. Example scenarios are listed in Table 19.

Those examples are from the previous troubleshooting research and focus on various performance diagnosis needs. In those scenarios, there are multiple or many system entities, such as servers, network equipments or VMs, and the analytics focuses on interactions among those entities, e.g. request paths, network communications. Table 19 lists a few of scenarios suitable for graph modeling. In this paper, we will show new use cases which use graph modeling to guide troubleshooting.

On the contrary, in the scenarios where single or a small number of entities are analyzed and/or there are a few interactions among entities, graph may not be the best choice. For instance, as shown in Table 20, detecting anomalies in time series monitoring data only considers individual metrics rather than the interactions among metrics or machines, therefore graph modeling is not suitable in such scenario. Similarly, in the troubleshooting practice where statistical correlations among metrics of servers or services are explored, it is difficult to fit it in graph modeling because the actual interactions among entities are ignored.

Table 19: Scenarios suitable for graph modeling

Scenario	Entity	Interaction	Approach
Network Dependency	Servers and Network Equipments	TCP/IP Packet Exchange	Sherlock [38] Orion [59]
Service Dependency	Web Services	Web Requests	Project5 [23] E2EProf [21]
Request Tracking	Services, Functions Processes,Threads	RPC Request/Response Function Call/Return	vPath [169] CAG [195]
VM Migration	VMs, Hosts	Migrations	CCM [109]
Machine/VM Communication	VMs, Machines	Network Communications	Net-Cohort [94]
Monitoring Data Summarization	Local Value, Aggregate Value	Aggregation Paths	SDIMS [193] Astrolabe [179]

5.2.2 Interaction Graph as Guidance

In large scale troubleshooting, one of the most critical tasks is to locate entities which cause the performance issue, for instance, the bottle-neck service, the failed disk and

Table 20: Scenarios unsuitable for graph modeling

Scenario	Entity	Interaction	Approach
Anomaly Detection in Time Series	Individual Monitoring Metrics	N/A	MASF [48], WISE [143]
Correlating System Metrics	Servers and Network Equipments	N/A	Pinpoint [110], Fingerprint [43]
Comparing System Behaviors	Hardware/Software Components	N/A	Kahuna [170], PeerWatch [105]

the out-lier monitoring metrics. Hence, we propose the guidance as a mechanism consists of a graph builder and a set of graph analysis functions which can dynamically, and automatically analyze distributed systems and output a subset of graph entities which are relevant to the root causes. Figure 43 illustrates a two-phase guidance mechanism. In the graph construction phase, VFocus collects metric data from all monitored entities in the system and builds interaction graphs on-line, at real-time. The interaction graphs are continuously updated to reflect the latest activities in the system. The second phase is graph analytics in which VFocus provides primitive operations to filter out the entities relevant to the performance issue. We will discuss the operations in detail in Section 5.3.4.

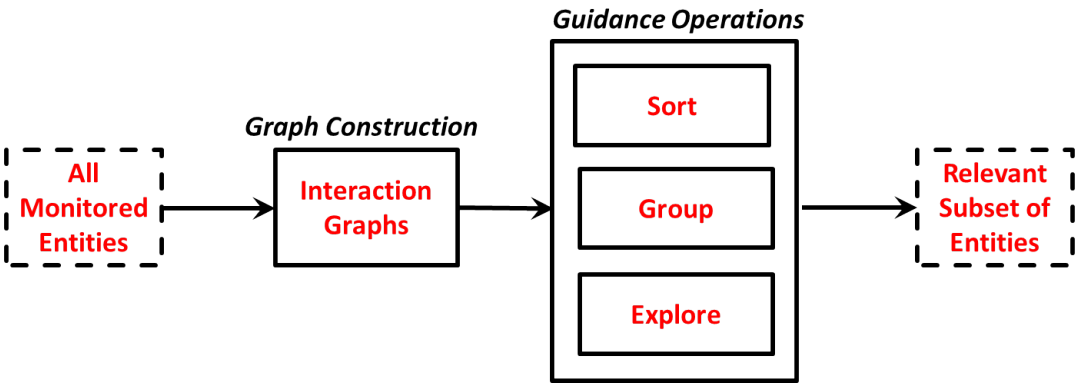

Figure 43: Graph-based Guidance Illustration

5.3 VFocus Design and Implementation

Though it is an integrated enhancement component of VScope, VFocus can be described as a system for tracking and analyzing user-definable interactions among components in data centers. If you look at VFocus as a black box, the input are monitoring metrics, the vertex, edge, and attribute specifications. The output are runtime graph snapshots taken on real time, and according analysis results. VFocus serves as the guidance component in VScope software stack (shown in Chapter III, Figure 21). It also provides extensions to VScope's *Scope* operation.

5.3.1 Interaction Graph and Snapshot

Interaction graph is one of VFocus' basic data structures. In an interaction graph, vertices are entities which VFocus is monitoring, and an edge between vertices describes an interaction between the two entities. The entities can be physical ones, such as physical machines or virtual machines in data centers. They can also be logical, for instance different types of the latency metrics. The interaction graph examples with physical and logical entities are illustrated in Figure 44 and Figure 45, respectively

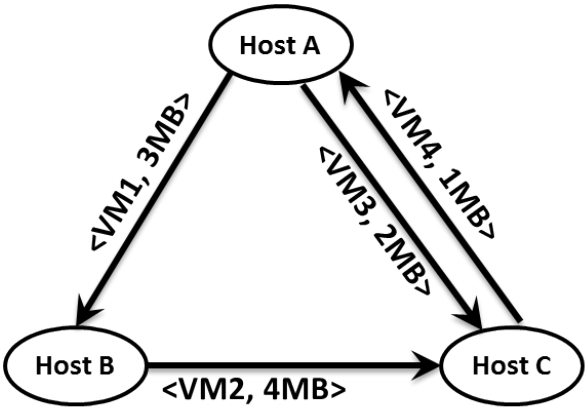

Figure 44: Illustration of a Graph Describing Physical Interactions

The interaction graph in Figure 44 describes Virtual Machine(VM) migrations among physical hosts. We will use this type of graph in Section 5.4.1 to troubleshoot migration failures. The vertices are VMware ESXi servers. The edges have the

VM being migrated and the attributes associated with this migration, for instance the servers' active memory size. The direction of the edge indicates the source and destination of migration.

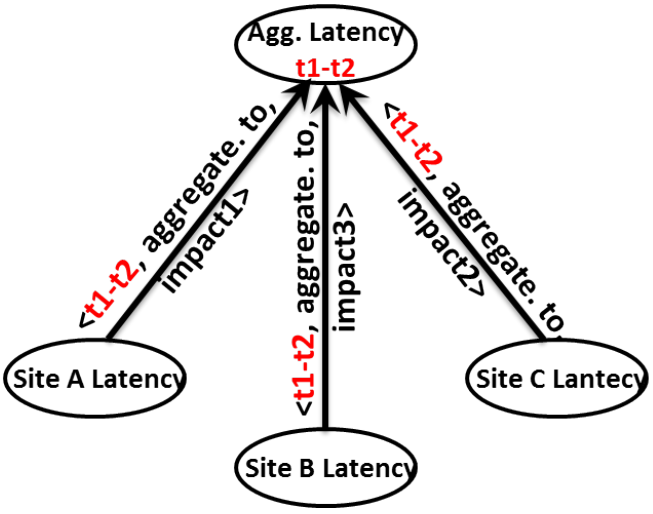

Figure 45: Illustration of a Graph Describing Logical Interactions

The interaction graph in Figure 45 describes the relationships between the aggregated latency and the individual latency measurements of a group of e-commerce websites. The aggregated latency is essentially a TP50 aggregation of all latency measurements from all the individual website's requests.The vertices are latencies of different websites. The edges describe the 'weight' of this website's contribution to the aggregated latency, the impact, which will be detailed in Section 5.4.2. We use this type of graph in Section 5.4.2 to find dominating latencies which contribute most to the aggregated latency.

A snapshot is simply an interaction graph with the timestamp when the graph is constructed. By taking snapshots of interaction graphs, VFocus can track the interactions evolving over time.

5.3.2 Graph Construction

VFocus uses DPG as the basic data collection and analysis infrastructure. As shown in Figure 46, VFocus typically creates a global DPG to collect interaction related

metrics from all the nodes in the monitored system. The DPG runs continuously through the troubleshooting process. Based on our experience in VScope and the experiments evaluating VFocus, a hierarchical DPG topology is enough to generate global snapshots of over 1000 nodes in a timely manner.

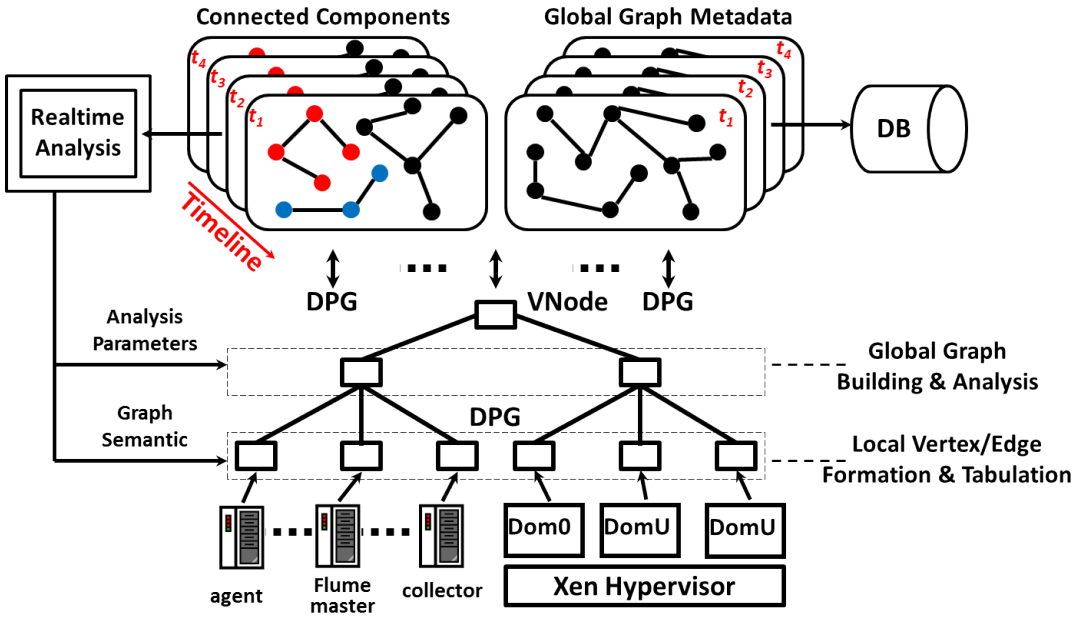

Figure 46: VFocus Architecture

VNodes on the monitored nodes collect metrics and build local graph data structures, including edge lists and vertex lists. The local data structures are then transmitted to intermediate VNodes for partial aggregation. The edge lists and vertex lists are merged and updated, forming the local aggregated graphs which are finally aggregated at the VMaster, generating a global graph. The global graphs are stored in memory of VMaster node for various analysis, and can be persisted to Database after a period of time. In our current implementation we focus on online troubleshooting, so graphs are discarded after it expires by configurable time window.

5.3.3 Graph Analysis Functions

VFocus provides basic graph analysis functions which are listed in Table 21. There are functions to calculate the basic properties of a graph, such as the number of

Table 21: Graph Analysis Functions

Function	Description	Graph Algorithm
Attribute Search	search a vertex/edge	DFS/BFS
Vertex Count	# of vertices	vertex vector
Edge Count	# of edges	edge vector
Get Attribute	get attribute	edge/vertex attribute
Degree Count	degree of a vertex	adjacency lists
Neighbor Analysis	direct/indirect neighbors	adjacency lists
Cluster Analysis	grouping vertices	connected components, cliques

Table 22: Guidance Operations and Scenarios

Operations	Analysis Used	Scenarios
sort	Degree Count, Get Attribute	VM Migration (in this Chapter), Dominator Analysis (in this Chapter)
group	Cluster/Clique Analysis	VM Clustering (in Chapter II)
explore	Neighbor Analysis	Naughty VM (in Chapter III), Culprit Region Server (Chapter III), Data Conflict (in this Chapter)

vertices and edges, degree of a specific vertex. In Section 5.4.2, we will present a use case which uses degree count to find candidate servers which are likely to have VM migration failures. There are also functions tracking the relationship among vertices such as neighbor analysis and cluster analysis. In Section 5.4.3, we will describe a use case which uses neighbor analysis to figure out the regionserver with data conflict issue.

5.3.4 Guidance Operations

VScope provides three guidance operations *sort*, *group* and *explore*. Table 22 shows the graph analysis functions which are used in each guidance operation.

Sort operation orders the attributes of all entities and takes top entities in the ordered list as relevant. For instance, in Section 5.4.2, we sort hosts by the degree (meaning number of migrations) of each vertex (meaning physical host), and pick the top hosts which have most migrations as relevant hosts which are likely to have migration failures.

Group operation puts strongly connected entities together as a group, and picks groups as relevant entities. Graph clustering algorithms can be used to identify closely related entities. For instance, in Chapter II, we use hierarchical clustering algorithm to cluster the VMs which has talked frequently to each other.

Explore operation finds the neighbors of a vertex at different distances (measured by number of hops). A typical example is to find which components have with each other. A good example is in in Chapter 5.4.3, where we use explore operation to track the data connectivities among HBase region servers, Flume agents and HBase clients, in order to troubleshoot the data conflict issue.

5.3.5 VFocus Implementation

We use igraph library [11] to implement graph structure and analytics. In igraph, graphs are represented as an edge list which stores all the edges each of which are described as a tuple of start and end vertices identifiers.

VFocus is implemented in two parts. The first part is a hierarchical DPG. Figure 47 illustrates the workflow of graph construction. At the leaf level, leaf VNodes collect monitoring data and transfer them into local edge lists. Partial aggregations at the parent VNodes need synchronization on the subset of leaf VNodes. The global graph is created in a centralized way after the root (VMaster) receiving partially aggregated graphs from all the parents. Most of the analysis on interaction graph are conducted after the global graph is constructed and stored in memory. The reasons are: (1) A graph may be reused by multiple analyses. (2) analyzing graph structure in memory has low computation time. (3) some analytics functions require global graph. However, there are several graph analysis functions which are implemented in a distributed manner, e.g. counting the degree of a vertex.

The second part of VFocus implementation is the adapter to transform monitoring metrics into edge lists at the leave VNodes. In our prototype, we have implemented

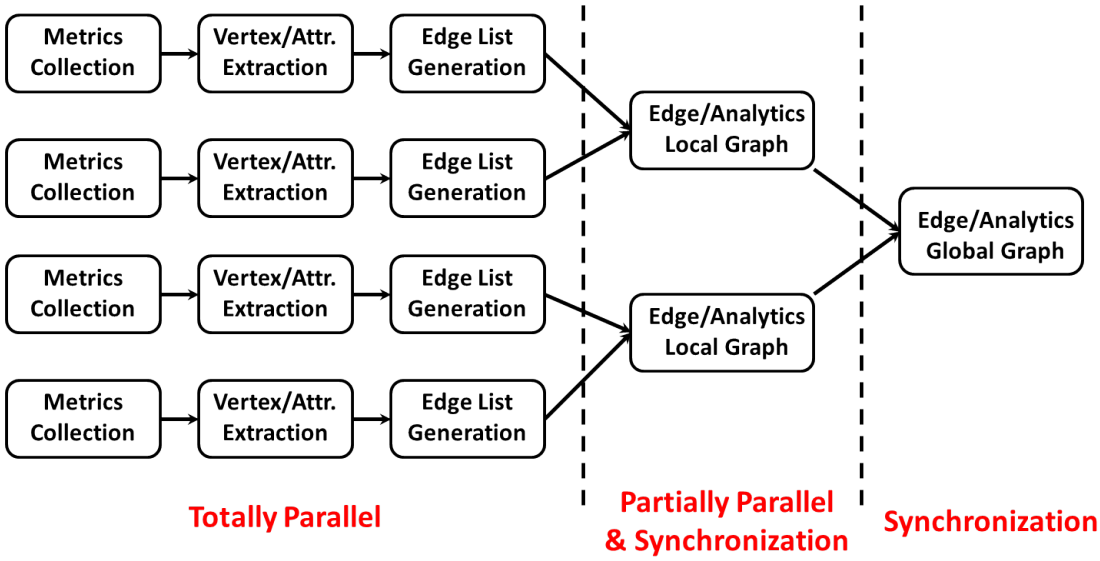

Figure 47: VFocus Workflow

three different adapters which will be described in detail in Section 5.4.

5.4 Use Cases

5.4.1 Dominator Analysis

The use case is related to a practical management issue from a real-world e-commerce service provider. A seller who wants to open an on-line store can use its services to easily build and manage his/her website. The service provider has data centers to host all the sellers' websites, and the key performance indicator for the performance of its service is the aggregated latency calculated by summarizing latencies of all the requests from all the sellers' websites, for instance, the p50 percentile of all the request latencies. The operation team keeps tracking the aggregated latency to make sure it is maintained at an acceptable low level. However, when it goes high, engineers need to figure out the reason. A typical question to ask first is which sellers have the most contributions to the increase. The challenges to answer this question come from (1) how to describe an individual seller's contribution to the aggregated latency. (2) how to track and find out the key contributor, i.e. the dominator.

The contribution, or impact, of one seller depends on two major factors. One is the number of requests the seller's website receives, and the other is the latencies of individual requests. As shown in Figure 48, we can deem each request to a seller as a circle. The larger the circle is and the more the circles are, the higher impact the seller has to the aggregated latency.

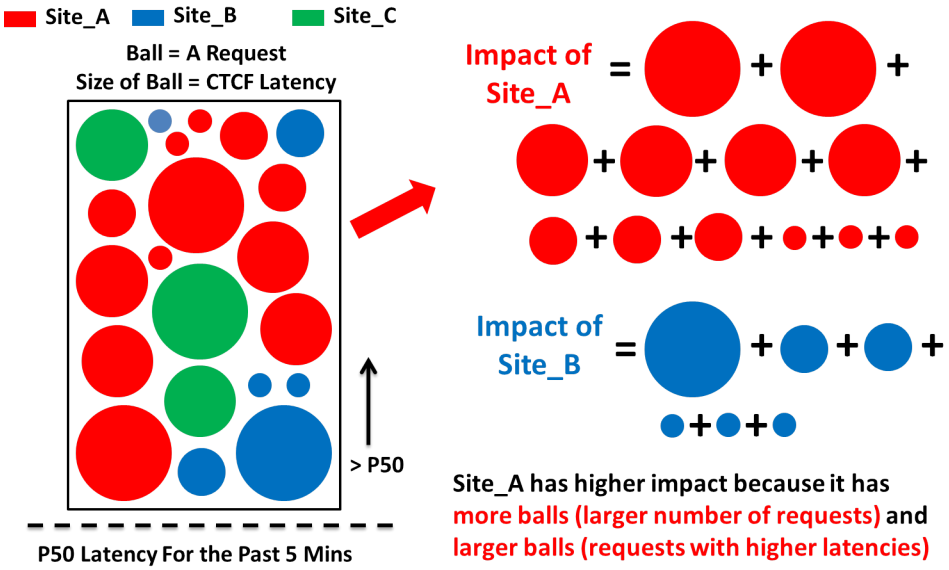

Figure 48: Illustration of Impact Attribute

Based on this reasoning, we define the impact of an entity (e.g. seller) i to the aggregated latency at a time t with a sampling rate of r minutes,

$$I_i = \sum_{j=0}^{n} L_{ij} \tag{9}$$

Where n is the total number of requests from entity i in the time period from t-r to t, L_{ij} is the latency of jth request from entity i .

Then we use VFocus to build a graph for tracking the relationship between each seller and the aggregated latency. An example graph was shown in Figure 45. The direction of the edges indicates "aggregated to" relationships, and the attribute of an edge is the impact of a seller calculated by Equation 9.

A snapshot of this interaction graph is created every r minutes (r is tunable). We

use the *sort* operation to order the edges by their impacts, and the top sellers on the list are the dominators. We also visualize the impact values of all the sellers at runtime in a heat map format where each square represents the impact of a specific seller (in row) at a specific time (in column), the brighter the color of the square the higher impact that seller has at that moment. Figure 49 illustrates the implementation of dominator analysis.

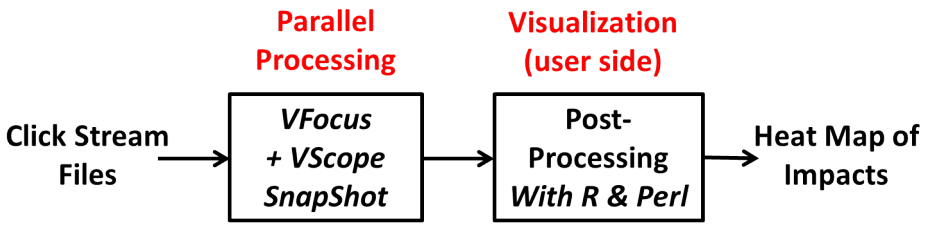

Figure 49: Dominator Analysis Workflow

Figure 5.4.1 shows an example of the heat map. We can easily identify a long term dominator (A53 in the heat map) who has top impact at almost all the time. At each specific time, one can also find the dominator at this time instance. We can also identify the sudden change of a seller in its impact, which indicates a sharp increase/decrease in number of requests for the latencies of requests.

Dominator analysis usecase shows that VFocus can be used to describe the logical, aggregation relationship between summarized measurement and the individual measurement. We can use the VFocus operations to identify the dominators to the aggregated measurement.

5.4.2 VM Migration Analysis

The purpose of this use case is to validate the usefulness of graph-based guidance by monitoring data from a real-world scenario. We have the traces of management operations in a virtualized data center in CERCS, Georgia Tech. The traces contain 256 servers running VMware vSphere Hypervisor (ESXi) v4.1, and hosting totally 1024 VMs. There are log files recording successful migrations. Each entry in the log has (1)

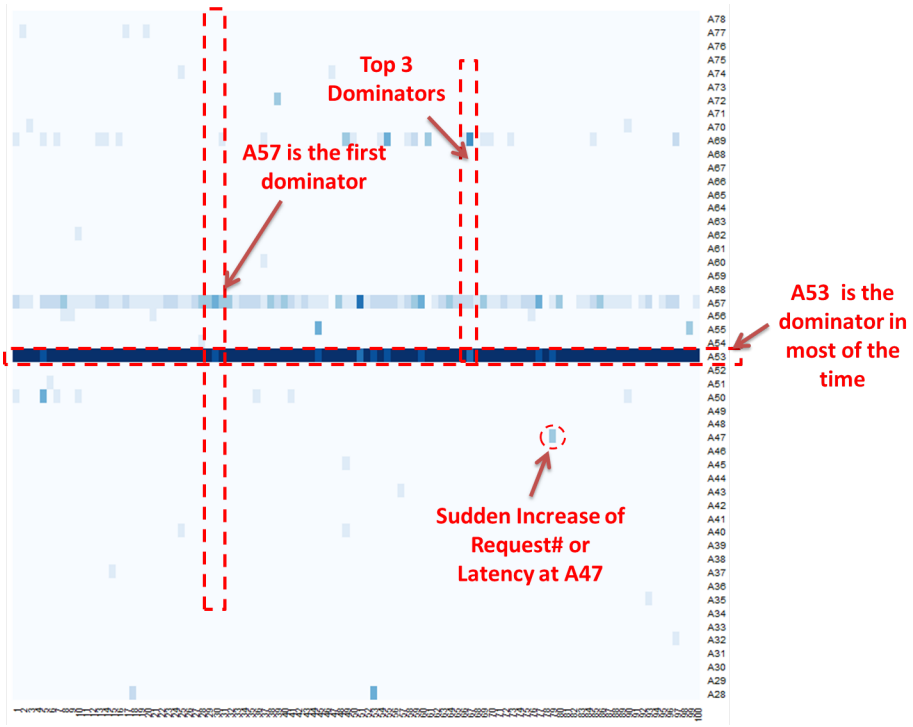

Figure 50: Dominator Analysis Illustration in Heat Map

starting time and ending time of the migration, (2) the name of VM being migrated, (3) the source host and destination host of the migration (3) VM CPU/Memory utilizations (4) source and destination hosts' CPU and memory utilizations. There are also log files recording the migration failures. Each entry contains (1) time when the failure happens (2) the name of the VM being migrated (3) the server hosting the VM (4) CPU and memory utilizations of VM and host and (5) the explanation for the issue. The logs record over 10000 migrations and over 1000 migration failures.

We use these log data to replay real time migration behavior. VFocus creates graph snapshots based on successful migration log data. We use a sliding window which takes a fixed number of migrations per window, and slide the window after every migration. The size of the sliding window is adjustable. In Section 5.5, we change the sliding window size to study VFocus's performance. In each graph, the vertices are the hosts and the edge is a migration linking the source host and destination host.

The logs are organized into 4 VCenters each of which is a management server in

Table 23: VFocus Troubleshooting Accuracy

VCenter	# overload error	# found overload error	# overload error / # total error	Accuracy
VC0	247	209	49%	85%
VC1	457	408	43%	89%
VC2	308	235	59%	76%
VC3	429	343	56%	80%
Overall	1441	1195	51%	83%

charge of a portion of 256 servers. To replay those logs, we initiate a DPG with 4 parents and each parent has one leaf VNode which takes the logs from a different VCenter log. A VNode reads the log file, extract time, source/destination hosts and VM names, and send the data to its parent. The parent will generate an edge list and according local graph. The local graphs are sent to VMaster to be aggregated into a global graph.

The methodology we used for validating VFocus is as follows. We take a snapshot of all the previous migrations at each time instance when a failure happens, which can be obtained from the migration failure logs, and we use the guidance operation *sort* on degrees of all the vertices (hosts) and rank the hosts in a descending order, i.e. the top hosts on the list are those with most migrations, and those hosts have more resource pressure than others, hence are more likely to have resource scarcity issue when new migration requests take place. To validate this idea, we then match the lists output at each failure point. If the actual host with migration failure is in the list. Then we consider it as a 'hit', because the VFocus method has successfully predict the failure node, otherwise we consider it as a 'miss'. In this research component we only study the failures are due to high resource pressure. This type of failure is indicated as Operation timed out in the explanation part of the failure record. We call this type of failure "overload error".

Table 23 shows the accuracy of VFocus approach. We can see in the table that, the overload errors are distributed evenly across VCenters (named vc0, vc1, vc2, vc3).

The overload error has a significant percentage of all the errors (over 50%). The overall accuracy is over 83%. The size of list we used is at most 45 hosts, which means, over the 256 hosts sorted by their degrees, we only pick the first 45. When we look at the histogram depicted in Figure 51 describing the percentage of the locations of hits (i.e. where on the list with 45 nodes has the hit happened), we can find that about 70% of hits happen within 30 hosts and nearly 40% is within 15 hosts. Therefore, with a high prediction accuracy, VFocus can find a host with overload errors within 15 hosts out of a search space with 256 hosts.

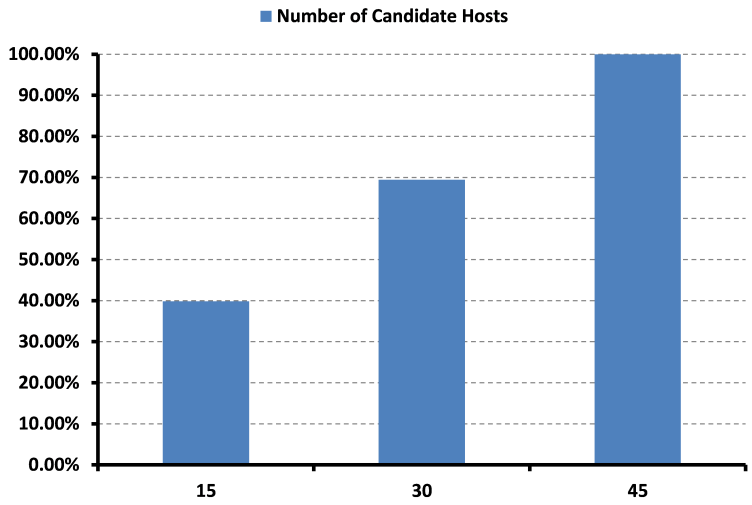

Figure 51: CDF of Hits

5.4.3 Data Conflict Analysis

One of the interesting issues we encountered in our stream analysis platform, which has been illustrated in Figure 18, is that some region servers in HBase may be overloaded due to application data conflict problem. The problem is illustrated in Figure 52. A Flume agent is writing data as key-value pairs into HBase, and at the same time, some other application such as a HBase client happens to write to the

same HBase table as well. If the row-key is well designed, the writing requests from both Flume and HBase would be distributed across all the region servers. However, one of the common pitfalls is using monotonically increasing row-keys (i.e., using a timestamp). In that situation, both Flume agent and HBase client are storing data (illustrated as red rectangle) to the same region server which becomes a hot spot of data conflict, i.e. they are writing to the same range of keys, causing resource (region server's memory heap) conflicts. Since that region server's memory are consumed much more quickly than other region servers, it will become the bottleneck of the HBase and slow down the end to end performance of the whole stream analysis platform.

This conflict is very difficult to detect because a HBase could be shared by different applications as an shared infrastructure. Therefore the data conflict could happen in an unexpected way because developers coming from different application may have no idea about how each other design his/her row key. Furthermore, the maintainer of the HBase may have absolutely no idea about any application's key assignment. HBase's Autosplit may work, but a threshold of the number of key should be predefined, and it is difficult to find a threshold because the key row designs of applications are dynamic and hard to predict.

In this usecase, we setup a testbed using GTStream [6] benchmark running FlumeNG [3]. In this experiment, we configure the row key generated by HBaseSink plugin, which is the Java class which Flume agents use to write key-value pairs to HBase, as the timestamp. We also run another HBase client using timestamp as the row key. The Flume agent and HBase client are writing to the HBase at the same time. The HBase has two region servers. We deploy VFocus DPG on all the nodes in the GTStream benchmark and track the TCP/IP packets exchanged between them.

In the network monitoring data collection, we use Libpcap [12] to sniff the network

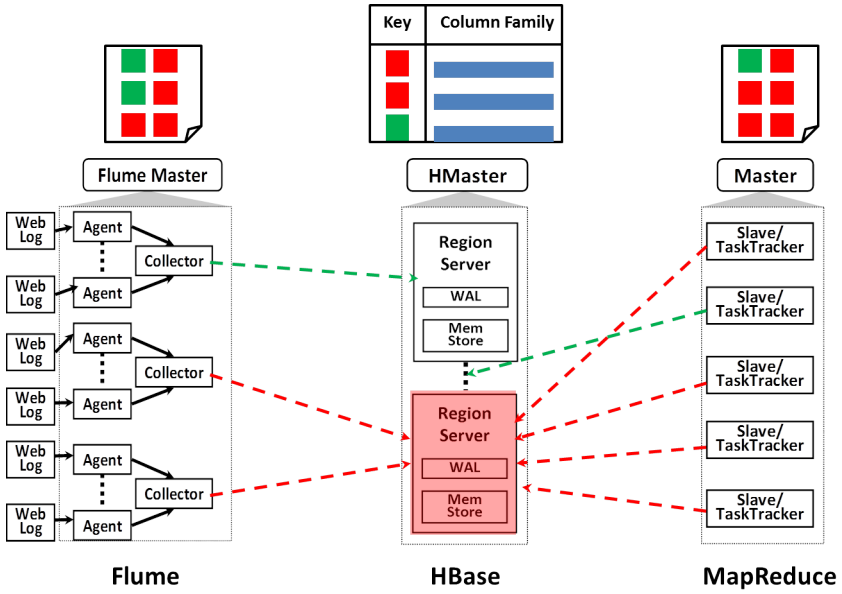

Figure 52: Illustration of Data Conflict Problem

traffic. Locally, each leaf VNode creates a local edge list with the source and destination host IPs as the source and destination vertices of an edge. We optimize the sniffing mechanism by filtering the packets that are irrelevant to the application. We ignore UDP packets and only record TCP packets sent to or received by the public ports used in HBase, Flume and HBase client.

VFocus also counts the number of packets transferred on that edge as its attribute. The graph snapshots are created per sampling period, and the duration of the sampling is tunable. We tested the VFocus's performance on different sampling durations in Section 3.3.

By using VFocus, we can track the interactions among all the nodes at runtime. Figure 53 shows the runtime snapshot of the partial graph between the among the two region servers, flume agent and the HBase client.The thickness of the edge represents the number of packets sent or received. We can tell that two edges on region server 2 is significantly thicker than the region server 1's, which indicates that this region server is overloaded by the unbalanced key-value pairs sent from the flume agent and the HBase client. In real world with hundreds of region servers and a number of

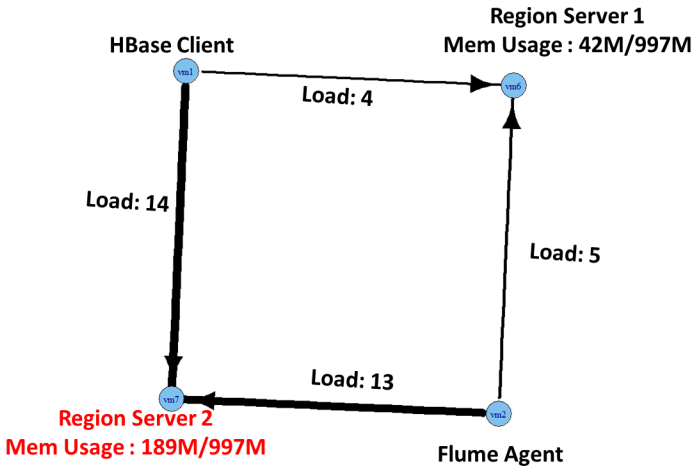

Figure 53: Illustration of an Interaction Snapshot

Flume agents and HBase client. The operator can use *explore* operation to figure out each region server's neighbors, i.e. the nodes with which each server interacts, and at the same time find the intensity of the interaction. Then he/she could figure out which region server is the potential hotspot with data conflicts.

This usecase validates that VFocus's effectiveness on a running system. It also shows the usefulness of the *explore* operation.

5.5 Performance Evaluation

5.5.1 Experimental Setup

Experiments are conducted on 100 VMs hosted by one physical server blades running Ubuntu Linux. The server has a 1TB SATA disk, 48GB Memory, and 16 CPUs (2.40GHz) and each VM has 2GB memory and at least 10G disk space. We deployed GTStream benchmark with 60 agent VMs, 10 regionserver VMs and 20 HDFS Data Node VMs. It is a fully distributed environment with FlumeNG deployed. We have 10 VMs serving as VFocus parents, HBase master, HDFS master and ZooKeeper servers.

5.5.2 VFocus Overhead

To test the overhead of VFocus on a VM, we run the network interaction tracking on VFocus and change the durations of sampling. At each change, we measure the CPU and Memory utilizations. The results are shown in Table 23. In addition we compare VFocus with the brute-force approach which uses TCPDump, turning on network sniffing all the time on all the packets. From the result table we can see that the overhead for VFocus is within 2% in CPU utilization while the memory consumption is neglectable. In contrast, brute-force approach consumes considerably higher CPU resources than VFocus. We believe VFocus achieves a better performance because the data collection part is implemented in a selective manner so it only looks at the packets belonging to the application. In real world scenarios, the ports, IP addresses are easily known to the application operators. Therefore it is trivial to make those optimizations especially when troubleshooting application performance issues. The CPU utilizations increase slightly as the duration increases because the network sniffing consumes more CPU cycles as it continuously runs for a longer period of time.

Table 24: VFocus v.s. Brute-force w.r.t. CPU & Memory Overheads

Sample Duration (seconds)	CPU Overhead	Memory Overhead
50	0.4%	0.2%
100	1.2%	0.2%
150	1.8%	0.2%
200	1.9%	0.2%
brute-force	35.5%	0.3%

We measure the application performance as the average end to end latency of the GTStream benchmark. We use the same method to measure the application performance change as in Chapter III. The interferences of VFocus and brute-force approach to the application are shown in Table 25. It shows that as the sampling duration increases, VFocus' interference to the application increases, while the interferences are

within 30% when the duration increases from 50 seconds to 200 seconds. By contrast, the always-on brute-force approach has a considerably higher interference at around 58%. To further study the breakdown of the overheads, we turned of the libpcap-based data connection function on all the Flume agents, and instead stored network connection metrics, which are pre-recorded in file using libpcap, in memory. Then we make VFocus read the memory and process data in the same way as we did when using libpcap. The third fourth column of Table 25 shows the Flume latency which is close to the baseline, indicating that the VFocus itself does not play the major role in the total overheads because it only incurs 7.06% slowdown, compared to 16.51% slowdown when libpcap-based monitoring is enabled.

Table 25: VFocus v.s. Brute-force w.r.t. Interference to Application

Sample Duration (seconds)	Latency (useconds)	Base Line (useconds)	Without Libpcap	Slowdown	Slowdown without Libpcap
50	4638210	3980879	3976479	16.51%	7.06%
100	4690591	-	-	17.83%	-
150	4890121	-	-	22.84%	-
200	5039372	-	-	26.59%	-
brute-force	6296441	-	-	58.17%	-

The experiment shows that VFocus has considerably lower overheads to the virtual machine and interference to the application compared to brute-force approach. By leveraging sampling and filtering optimizations, the overheads are reduced.

5.5.3 Graph Construction Performance

To evaluate VFocus' performance on graph construction, we conduct two experiments. In the first experiment, we test graph construction latency on the 100 VMs running GTStream benchmark. VFocus collects network communication data using Libpcap on agent VMs, aggregates the local graphs and generates a global graph. We vary the duration of sampling from 10 seconds to 150 seconds, Table 26 shows that the total time for generating a global graph is within 220000 useconds.

We also test the graph construction performance by generating VM migration graph in usecase 2 with different sliding window sizes, and compare two construction strategies, centralized and parallel. In the centralized strategy, we send all the successful log data to a central node and generate a global graph; while in the parallel strategy, we create the graph in a distributed way illustrated in Figure 47. The graph has up to 1000 edges. As shown in Figure 54, parallel construction outperforms centralized strategy and as the size of sliding window increases the construction time increases as well. However the maximum construction time is within 700 milliseconds, and the parallel construction strategy has much smaller increases as the window size increase than the parallel strategy's.

Table 26: Graph Construction Time on 100 VMs

Sample Duration (seconds)	Latency (useconds)
10	170943
50	174098
100	181646
150	213591

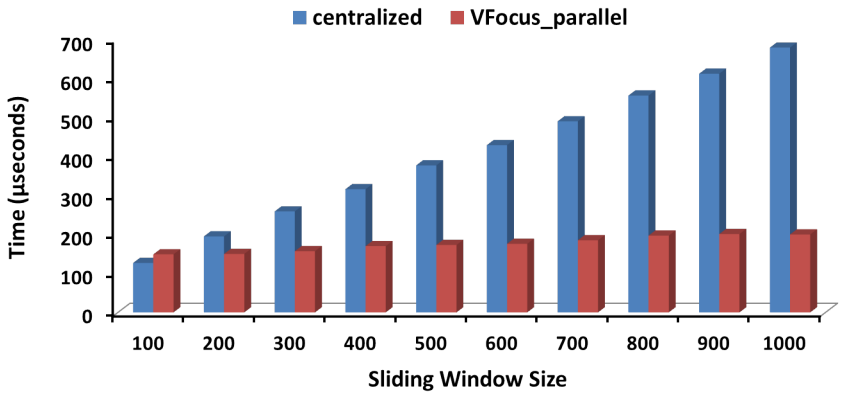

Figure 54: Graph Construction Time w.r.t Sliding Window Size

The results show than the graph construction is fast by using real data center monitoring traces, hence VFocus is capable of generating online, runtime interaction graphs. Parallel construction strategy used by VFocus is scalable and faster than the centralized strategy.

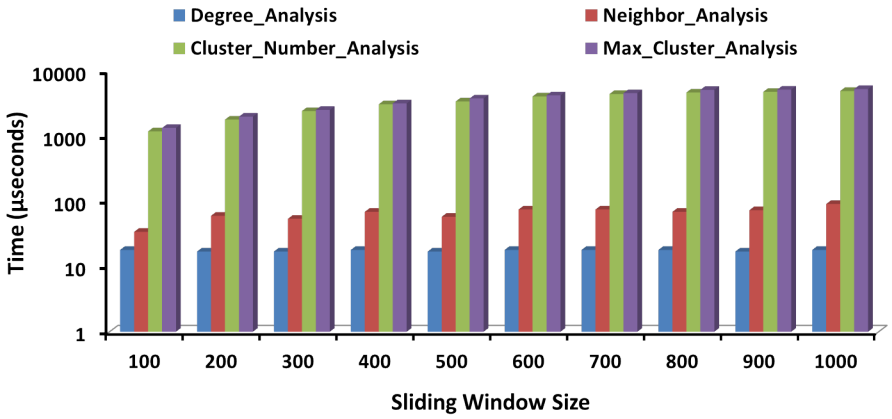

Figure 55: Graph Analysis Performance w.r.t Sliding Window Size

5.5.4 Graph Analysis Performance

We evaluate the VScope's performance on graph analysis by measuring the computation time of analysis listed in Table 21. Cluster analysis has two functions (1) Number of Cluster function that counts the number of cliques in the graph and (2) Max Cluster function that yields the largest cliques in the graph. The monitoring data is the successful VM migration logs and we measure computation times at different sliding window size.

As shown in Figure 55, the computation time is within 10000 microseconds when the sliding window size changes from 100 to 1000, which means for VM migration use case, the graph analysis can be conducted in a timely manner. As the sliding window enlarges, the time increases slightly, because all the computation are processed in memory, which make it suitable for real-time graph analysis. Among different analysis functions, the two cluster analysis functions have longer computation time because they have higher computation complexities than degree analysis and neighbor analysis.

CHAPTER VI

LITERATURE REVIEW

6.1 Terminology

A *data center* is a company's facility housing servers and associated components, such as networking equipment, storage systems, cooling systems, and power supplies. The number of servers in a data center can be large. Unofficially, it is estimated to a range from more than 1000 to over 1 million[1]. Data centers can be consolidated or/and non-consolidated, where consolidated data centers use virtualization technology to run virtual machines (VMs) on physical servers and deploy applications in VMs, while non-consolidated data centers deploy applications on physical servers. Hybrid data centers use both VMs and bare servers to host applications. Typically, the applications running in these systems are server-side applications, such as traditional web services, data analytics, and multimedia streaming services. They may belong to a single enterprise or to multiple parties, since servers or VMs can be rented to customers as managed data centers and/or cloud services.

Depending on management needs and service objectives, *performance* can be defined at different levels of abstraction — application, VM, OS, hardware, and from different viewpoints — application user, infrastructure provider, etc.. Performance can also be described for an individual server (e.g., the server's CPU utilization, memory utilization) or for a set of servers as aggregate metrics (e.g., ranked power usage, application level request latency of a web service hosted by a number of servers).

[1]Companies like Google don't officially reveal the size of their data centers. Estimates are based on calculations of data center energy consumption, etc.

However, in daily data center operations, there are typically a small number of metrics of importance to a company's business interests. These metrics, termed *key performance indicators — KPIs —* are commonly used as measurements describing high-level performance objectives. Each data center management scenario may have a unique set of KPIs.

We define *performance problems* [184] as violations of KPIs and/or issues (failures, errors, resource contention, or other abnormalities) that contribute to KPI violations. In practice, for each user and provider, there is a relatively small number of KPIs, but there is an extensive set of low-level measurements for each problem, and operator definitions/perceptions of such problems vary.

Performance problems occur due to a large variety of root causes. Faults can occur at each level of the data center — from transient or permanent hardware faults, to unforeseen component interactions, to software misconfiguration or bugs. This bibliography focuses on only those methods and systems that address *performance troubleshooting*, which can be defined as a procedure that (1) detects a performance problem, (2) diagnoses the problem to the extent needed to remedy it, mitigate its effects, or even determine its root cause, and (3) applies remediation activities to restore the system back into correctly operating state.

6.2 Methodology

Performance troubleshooting in enterprise date centers typically involves the following phases.

1. Detection is the phase that identifies violations of KPIs or anomalies that may lead to KPI violations. The outcomes of detection includes the time(s) when the problem is detected and the metrics or KPIs related to the problem. With KPIs that denote end-to-end performance, detection of KPI violations is often too late because performance degradation has already affected user experiences. These violations also

typically lead to loss in revenue and/or system downtime. Furthermore, KPIs are hard to define in some circumstances, particularly in public clouds, as a cloud users' KPI may be unknown. Research has responded to these issues by moving from directly measuring KPI violations to detecting candidate anomalies before such violations occur, and as a result, anomaly detection has become an important area of research in performance troubleshooting. We will discuss this research in Section 6.3.

2. Diagnosis is the phase in which KPI violations or anomalies are analyzed to determine their causes or/and suggest solutions. One of the outcomes of diagnosis phase is the *location* of the problem, which enables subsequent mitigation or remediation. However, identifying root-causes and fixing them typically require human involvement and/or the use of sophisticated problem-solving tools that typically require domain expertise to make intelligent inferences.

3. Remediation or mitigation is the phase of taking actions to recover the system to normal state or to prevent it staying in an abnormal state. The result of remediation can validate the detection and diagnosis phases, making it possible to create a 'closed loop' for automated performance troubleshooting.

Though there is a few research in which there is no clear border between detection and diagnosis phases, we follow the organization suggested above when categorizing existing literature, for clarity sake. We also classify the systems and infrastructures that support performance troubleshooting as follows.

1. Monitoring systems collect and aggregate metrics from the systems they observe. Monitoring systems aim to provide a global view. They typically compile summary statistics to facilitate continuous monitoring in large scale applications and systems.

2. Tracing systems aims to provide detailed information by collecting targeted events from the systems being observed, such as the messages being sent and received, system calls executed, *etc.* Tracing systems may also track certain system behaviors, like the call patterns or service call paths present in distributed applications. Tracing

can impose considerable overheads on target systems, with large trace logs stored in files. Tracing has recently gained additional importance due to the widespread use of instrumentation, logging user and/or system events by web companies able to monetize such data.

3. Analytics systems support analysis and queries, based on monitoring and/or tracing data acquired about the targets under observation. They may operate online — as their target applications run, offline — based on previously collected monitoring or log data, or both, sometimes also termed as operating on 'data in motion' vs. 'data at rest'.

6.3 Detection

Problem detection algorithms can be classified into two categories: (1) *reactive approaches* that detect anomalies after they occur, and (2) *proactive approaches* that predict impending anomalies when the system state is still normal. The reactive approach does not incur any *prevention* cost but can lead to prolonged service downtime — which can be prohibitive for real-time applications like online web logs, live sensor stream processing [186], and many business applications [80]. In contrast, the proactive approach offers better responsiveness, but at the risk of imposing continuous overhead for prediction and possible penalties for inaccurate detection of anomalies — false alarms. We will follow this categorization when detailing existing approaches in Section 6.3.2 and Section 6.3.1.

6.3.1 Reactive Detection

Reactive approaches aim to detect problematic states of data center systems or applications. Data center users are concerned with external states, expressed by KPIs. Data center operators are concerned with internal states — their daily jobs being to handle abnormal internal states. However, abnormal states are difficult to describe with simple metrics and associated thresholds. This is due, in part, to system scale

and complexity; but a more important reason is that performance problems in the data center may only manifest at application level. An instance reported in [110] is a web site that correctly responds to pings, HTTP requests, and returns valid HTML; yet, at the application-level, returns only partially filled web pages or incorrect page content. These problems are difficult for data center operators to detect because they have limited knowledge about applications. These difficulties are compounded by the diversity of data center software and hardware provided and maintained by multiple vendors or organizations. Problem detection methods are designed from many different perspectives. From the knowledge base perspective, MASF (Multivariate Adaptive Statistical Filtering) [48] and WISE [143] are *threshold-based* algorithms setting up thresholds of metric values along with the time pattern used for monitoring. They detect performance problems as threshold violations on individual observations [48] or on a series of observations [143]. Threshold-based approaches typically cater for seasonal variations in workload, *e.g.*, heavier load during the week and lighter load over the weekend, by using different thresholds for different seasons. The main challenge with these techniques is selecting a threshold that maximizes true positives while minimizing false positives.

Pinpoint [110], Console Log Analysis [192], SPA (System Performance Advisor) [93] and the profile-driven model [168] are *modeling-based* approaches that use machine learning models [110, 192], time-series models [93] and queuing models [93, 168] to describe computer system performance, where performance problems are recognized by deviations from these models. Model-based approaches are well-suited for detecting application-level problems. However, building models requires an in-depth understanding of the system.

Flow Intensity [99], PeerWatch [105], Reference-Execution [164], Metric-Correlation Models [102] and Invariants Mining [132] are *correlation-based* approaches. Correlation-based approaches learn normal behavior by analyzing historical data to

automatically discover correlations between metrics that are stable over time. They assume that performance problems break these correlations. Care needs to be taken when selecting thresholds for determining whether a correlation is significant — if the thresholds are too low, these approaches may discover spurious relationships which result in increased false positives.

Instead of detecting violations on metric values, as all aforementioned approaches do, EbAT [188, 191, 183, 189, 123], NMI(Normalized Mutual Information) [101], and Parametric Mixture Distribution [151] use *information and probability theories* to detect anomalies in metric distributions. For instance, EbAT aggregates multiple metrics into a random variable, and tracks their distribution patterns at runtime. The anomalies are detected via identification of outliers in the distribution patterns. There is also research on post-detection, concerned with reducing false positives or false alarms and/or to prioritize the severity of anomalies. Examples include the anomaly-ranking approaches described in [180, 181, 100].

6.3.2 Proactive Detection

A shortcoming of reactive approaches is that they cannot prevent performance problems from occurring. Proactive approaches usually assume that there are existing measurements to describe normal or abnormal states of the system (KPIs or specific metrics and associated thresholds/patterns). They raise alarms when the system in still in normal state by looking at the predefined measurements, predicting when the problem will happen and on which machine.

[171] investigated the predictability of real-world systems and concluded that anomaly prediction approaches can be applied to them with high accuracy and significant lead time. [87] employed Bayesian classification methods and Markov models to predict impending anomalies in distributed data streaming applications online. ALERT [172] used an adaptive scheme to adjust the prediction models in

different workload contexts, addressing the dynamism in date center environment. Prepare [173] presents an anomaly prediction approach for virtualized cloud. [60] proposed a integrated framework of measurement and system modeling techniques to detect application performance changes and differentiating performance anomalies and workload changes.

6.4 Diagnosis

6.4.1 Dependency Inference

An important topic in diagnosis is to understand the spatial or temporal relationships between software or hardware components in enterprise data centers. *Dependency inference* analyzes relationships between interconnected components for problem troubleshooting, particularly when localizing the causes of problems that propagate across a distributed system. Specific instances of insights sought by dependency inference include service dependencies, request or call paths and transaction tracking through a distributed system.

Dependencies can be described at different levels, leading to different inference approaches. Orion [59], Project5 [23], Sherlock [38], ADD (Active Dependency Discovery) [46], and E2EProf [21] infer machine level dependencies. For request level diagnosis, Spectroscope [161] is a diagnosis approach that identifies and ranks changes in the flow and/or the timing of request processing, by comparing requests flows from two executions of the same application. X-ray [37] diagnoses the root causes of performance problems by instrumenting binaries as applications execute, and by using dynamic information flow tracking to estimate the likelihood that a block was executed due to each potential root cause. X-ray also highlights performance differences between two similar activities by differentially comparing their request execution paths. State-transition-based Transaction Tracking [27], CAG [195], vPath [169], and NetMedic [104] track process or thread level dependencies.

Active Probing [160] uses an incremental approach to infer service dependencies via active probing. Pranaali [117] also infers relationships and partitions between subcomponents of services. MonitorRank [112] periodically generates service dependencies which are then leveraged by unsupervised machine learning models for suggesting root cause candidates.

Researchers use a variety of techniques for inference. Orion [59], Project5 [23], and E2EProf [21] use network traffic and signal processing methods to infer dependencies without detailed knowledge about the application being observed, so that they can operate in the 'black box' settings experienced in virtualized systems. Sherlock [38] and NetMedic [104] leverage application-level knowledge such as configuration information and historical failure/success records, along with network traffic information, to infer dependencies. CAG [195] and vPath [169] use system- or kernel-level traces to infer dependencies, while X-ray [37] uses binary instrumentation to infer application-level dependencies.

State-transition-based Transaction Tracking [27] uses Markov Chain models along with the ARM instrumentation metrics to infer the dependencies of transaction footprints. ADD (Active Dependency Discovery) [46] finds dynamic dependencies by systematically perturbing the system, which is also used in the Bayesian modeling based approach in [117] to first partition a larger scale system into different state spaces, each of which can then be investigated separately.

6.4.2 Correlation Analysis

Correlation analysis explores the correlations between the observed metrics and the problematic/normal behaviors of data centers, or it considers correlations between metrics to characterize the patterns of the system, (*e.g.*, clustering metrics in some normal system state to construct a model of such states). Most correlation analysis approaches use machine learning techniques to correlate metrics to known problems.

Signature [61, 62], HPD (Heath Problem Detection) [18], Symptom-Matching [45], Fingerprint [43], iManage [119], Pinpoint [57], and Trace-based Problem Diagnosis [194] correlate SLA violations or known problems to system-level metrics. [19] correlates system changes to the failure or success symptoms of the system. CAL [58] correlates a request path to its failure or success states using decision tree methods. Draco [107] uses Bayesian analysis to localize chronic problems by comparing the distribution of features in successful and failed requests. [192] and Fa [68] build correlation models among monitoring data in normal state by using clustering methods.

6.4.3 Similarity Analysis

Peer Comparison [106], Ganesha [152], Kahuna [170], and PeerWatch [105] detect and localize problems by comparing behaviors of peer machines or peer software components, under the assumption that in normal state the peers should perform similarly. Hence, the outliers deviating from the similarity are the problematic nodes or components. Peer-comparison approaches assume that the majority of peers in the system are fault-free.

6.4.4 Detection vs. Diagnosis

Detection and diagnosis are typically applied to two different phases of performance troubleshooting procedure. For instance, detailed diagnosis of root-causes usually takes place after alarms are raised by anomaly detection. Based on the nature of the their respective timings, a detection approach should primarily consider responsiveness while diagnosis should emphasize comprehensiveness and detailedness. That's why we can find that current detection methods are usually on-line and apply to small volume, real-time monitoring metrics while diagnosis approaches are usually off-line and analyze large volumes of historical monitoring data.

However, above differences are not decisive. Some diagnosis approaches such as Kahuna [170] can do on-line diagnosis while some detection approaches like [192] need

off-line analysis on history data to build a model beforehand for real-time anomaly detection. In this sense, the difference between detection and diagnosis is blurring.

6.5 Supporting Infrastructures

6.5.1 Monitoring Infrastructures

Monitoring systems are the foundation for performance troubleshooting in data centers — their purpose being to keep track of the 'health' status of the whole data center and/or the end-to-end performance of the applications. As a result, scalability, on-line operation, and low overheads that minimally perturb running applications are the dominant design criteria.

Monitoring infrastructures can be categorized into state monitoring vs. aggregation systems. Ganglia [138], Nagios [78], REMO [142], Query [157], CoMon [153], and foTrack [196] are *state monitoring systems* that are able to track overall system states in large scale data centers, ranging from thousands of nodes (as in Ganglia) to 60,000+ nodes (as in Query). These systems typically use simple analysis functions like thresholding, and there has been research [142, 196] to optimize analysis algorithms to reduce false positives and overheads in terms of resource consumption for monitoring. *Aggregation systems* collect and summarize large volumes of monitoring data distributed among data center nodes, to yield a global summary of system health. Astrolabe [179], SDIMS [193], Moara [114], San Fermin [52], and Network Imprecision [98] are representative systems in this category.

6.5.2 Tracing Infrastructures

The typical purpose of tracing infrastructures is to support the diagnosis phase of performance troubleshooting. They provide mechanisms to trace events both along (*i.e.*, horizontally) and across (*i.e.*, vertically) the software stack in data center systems. Data may be collected at hypervisor, kernel, middleware, and application levels, and such data typically consists of extensive trace logs that are not typically seen in the

monitoring systems described in Section 6.5.1 because of the potentially high overheads or perturbation associated with their collection and use. Research in this space, therefore, is often concerned with (1) how to reduce tracing overhead and perturbation, (2) how to easily and effectively instrument systems and applications, and (3) how to carry out on-line tracing at acceptable cost.

Fay [73], Chopstix [42], Dapper [165], and GWP (Google-Wide Profiling) [156] use sampling techniques to trace its large scale data center on-line with high effectiveness in problem detection and diagnosis.

Whodunit [54], DARC [176], Stardust [174], X-Trace [75], D^3S [128], and Chirp [30] provide systems or programming tools to instrument data center applications or systems for tracing and querying their runtime states. SysProf [20] uses kernel level instrumentation to realize fine-grained, low-overhead tracing. Pip [158] is an instrumentation framework that allows programmers to embed expectations about a system's communications structure, timing and resource consumption. Pip detects problems by comparing actual behavior against expected behavior.

6.5.3 Analytics Infrastructures

Analytics infrastructures support performance troubleshooting by making it easy to apply various analytics functions to monitoring metrics in data center. Like data mining systems, their focus is data analysis rather than the data collection actions taken by tracking and monitoring infrastructures though some of these analytics systems, however, have associated data monitoring and tracing functions. Magpie [39] is a system for tracing and also clustering requests to learn about interaction patterns in the system being observed. SelfTalk [83] is a query system supporting various queries useful for understanding system behaviors of multi-tier web applications. Monalytics [120, 187, 190] combines monitoring and existing analytics approaches (detection,

diagnosis approaches) to offer a flexible architecture by which a wide range of analytics can be applied to various monitoring metrics/traces on any node in data centers. Monalytics was built on the vManage [116] architecture designed for managing virtualized data centers. VScope [185] is a flexible middleware for troubleshooting performance problems in large scale real-time big data applications.

6.6 Remediation

As stated earlier, a wide variety of techniques and approaches are used to mitigate or remedy problems. Below, we only review the representative recent work focused on data center systems and applications.

The TCP splice-based web server [136], SLACH [200], and the data recovery system [24] study novel system designs to support fault tolerance in enterprise data center applications. NAP [41], Policy Refinement [84], QoS-aware fault tolerant middleware [199], and [201] specialize in planning and optimizing recovery policies to best improve system performance. RobustStore [47], FATE and DESTINI [89], RFD analysis models [133], and the analytic queuing models used in [162] are tools and models that analyze the effects of recovery actions on applications.

Recovery actions most commonly use rebooting for failure recovery or performance remediation, with improvements sought by methods like Micro-reboot [50], JAGR [51], RETRO [113], Stochastic Reward Nets(SRNs), and Software Rejuvenation [178].

In recent big-data frameworks, such as MapReduce, performance problems are addressed by rescheduling slow tasks, and intelligent data placement. Mantri [29] uses statistical tests to detect 'straggler' tasks caused by data skew, network congestion and machine failures in large-scale MapReduce jobs. Mantri takes the appropriate recovery action, e.g., task re-execution, based on the underlying cause. Scarlett [28] tackles the file system performance problems in MapReduce clusters by predicting the

data 'hotspots' and replicating data blocks based on their popularity. In virtualized data center scenario, Net-cohort [94] leverages clustering techniques to discover VM cohort which is a group of VMs working closely to each other. By migrating VMs, which belong to the same cohort, to the same physical host, Net-cohort can effectively relieve the resource bottleneck (e.g. cross-rack bandwidth contention) and thereby improve the performance of big data applications running in the data center. There is much valuable, previous and ongoing research in high reliability systems obtained through redundancy and replication [92], which in turn builds on extensive prior work in reliability study. In data center systems' 'scale-out' infrastructures, this has led to the common use of data replication, fail-over servers, quorums, and the many associated methods for fault tolerant, high availability operation. We do not review such work here, pointing the reader at previous summaries for research like [88].

CHAPTER VII

CONCLUSION AND FUTURE WORK

7.1 Conclusion

We present the Monalytics software architecture for integrating monitoring and analytics in large scale data centers, with flexibility for supporting a variety of analytics functions. We introduce *pro re nata*(PRN) methods and experimental evaluations are carried out with a Monalytics software prototype implemented in small scale data center running three tier enterprise applications and Hadoop codes. Results clearly show the importance of using PRN, along with the ability of the current Monalytics prototype to support the multiple and sophisticated monitoring/analysis functions required by two realistic use cases. We contribute novel analytical formulations modeling DCG's effects on both the performance and the capital costs of monitoring/analysis, with extensive analytical evaluations in large scale.

Based on Monalytics architecture and analytical models, we design and implement VScope, a flexible, agile monitoring and analysis system for troubleshooting real-time multi-tier applications. Its dynamically created DPG processing overlays combine the capture of monitoring metrics with their on-line processing, (i) for responsive, low overhead problem detection and tracking, and (ii) to guide heavier weight diagnosis entailing detailed querying of potential problem sources. With 'guidance' reducing the costs of diagnosis, VScope can operate efficiently at the scales of typical data center applications and at the speeds commensurate with those applications' timescales of problem development. The dissertation provides evidence of this fact with a real-time, multi-tier web log analysis application.

We also present EbAT which is an automated online detection framework for

anomaly identification and tracking in data center systems. It does not require human intervention or use predefined anomaly models/rules. To deal with the complexity and scale of monitoring, EbAT uses efficient m-events to aggregate different levels of metrics in clouds, leverages entropy-based metric distributions, time series diagnosis methods to detect anomalies at runtime, and zoom in detection to focus on possible areas of causes.

7.2 Future Work

Future research work includes, but not limited to, following directions:

- `System Analytics Service`. One possible future research direction is to transform VScope which is an software tool for data center operators' internal use to a cloud service providing monitoring and analytics capabilities to different outside applications. Monitoring/analysis as a service is an emerging research area. Meng [141] has done extensive research on Monitoring-as-a-Service in Cloud. Amazon has CloudWatch [1], which is a monitoring service with APIs to monitor and analyze the cloud resources used by EC2 customers. The future challenges include how to integrate more analytics capabilities into the cloud services, and how to realize real-time streaming analysis on large volumes of monitoring data.

- `Automatic Management Planning`. As data centers scales out, the cost and performance of the management infrastructure increase accordingly. Previous research [167] has shown the considerable management cost in virtualized data centers. Controlling and optimizing the resources for management purposes can only become more critical as data centers move to exascale. In monalytics research, we have shown that, different management strategies can yield significantly different capital cost and performance. In future research, we will focus on how to design a control plane which can realize management strategies

by deploying management infrastructure to data center accordingly, and more interestingly, dynamically changing the management layout as the data center and management needs change.

- `Cloud Resource Management`. VScope is a tool to collect and analyze monitoring data for troubleshooting purposes. Future research will focus on extending VScope's capabilities for resource management purposes, such as Virtual Machine allocation and migration management. In this scenario, VScope can still collect monitoring data and analyze them, but the challenges include (1) how to expose a set of APIs that can be seamlessly integrated with the resource management operations and facilities in large scale systems [198]. (2) what analysis can be realized in VScope to facilitate cloud resource and power management [81].

- `Cloud Security`. As more and more critical applications move to the cloud, security becomes one of most important issues. VScope's real time analytic capabilities can be used in security applications. By collecting and analyzing network traffics, VScope can use network anomaly detection algorithms to find out potential attacks or security breaches. One interesting research direction is to monitor the network interactions at runtime to dynamically learn the normal data center behavior, and adjust the normal behavior as new workload patterns happen. The normal pattern can be used to identify abnormal behaviors.

REFERENCES

[1] "Amazon cloudwatch," http://aws.amazon.com/cloudwatch/.

[2] "Amazon ec2 website." `http://aws.amazon.com/ec2/`.

[3] "Apache flume." `http://flume.apache.org/`.

[4] "Apache hadoop," http://hadoop.apache.org/.

[5] "f_1 score in wikipedia." `http://en.wikipedia.org/wiki/F-score`.

[6] "Gtstream." `https://github.com/chengweiwang/GTStream`.

[7] "Hp ilo," http://h18000.www1.hp.com/products/servers/management /remotemgmt.html.

[8] "Hp systems insight manager." `http://www.hp.com/go/sim`.

[9] "Ibm systems director," http://www-03.ibm.com/systems/software/director/.

[10] "Ibm tivoli." `http://www.ibm.com/tivoli`.

[11] "igraph library." `http://igraph.sourceforge.net/`.

[12] "Libpcap." `http://www.tcpdump.org/`.

[13] "Nimsoft website." `http://www.nimsoft.com`.

[14] "Notes from hp production team," Personal Correspondence, Sep. 2010.

[15] "opencirrus website." `https://opencirrus.org/`.

[16] "precision and recall in wikipedia." `http://en.wikipedia.org/wiki/ Precisionandrecall`.

[17] Abadi, D., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J., Lindner, W., Maskey, A., Rasin, A., and Ryvkina, E., "The Design of the Borealis Stream Processing Engine," *Second Biennial Conference on Innovative Data Systems Research*, 2005.

[18] Agarwal, M. K., Gupta, M., Mann, V., Sachindran, N., Anerousis, N., and Mummert, L. B., "Problem Determination in Enterprise Middleware Systems using Change Point Correlation of Time Series Data," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2006.

[19] Agarwal, M. and Madduri, V., "Correlating Failures with Asynchronous Changes for Root Cause Analysis in Enterprise Environments," in *IEEE Conference on Dependable Systems and Networks (DSN)*, 2010.

[20] Agarwala, S. and Schwan, K., "SysProf: Online Distributed Behavior Diagnosis through Fine-Grain System Monitoring," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2006.

[21] Agarwala, S., Alegre, F., Schwan, K., and Mehalingham, J., "E2EProf: Automated End-to-End Performance Management for Enterprise Systems," in *IEEE Conference on Dependable Systems and Networks (DSN)*, 2007.

[22] Agarwala, S., Chen, Y., Milojicic, D., and Schwan, K., "QMON: QoS- and Utility-Aware Monitoring in Enterprise Systems," in *ACM International Conference on Automatic Computing (ICAC)*, 2006.

[23] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A., "Performance debugging for distributed system of black boxes," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[24] Akkus, I. and Goel, A., "Data Recovery for Web Applications," in *IEEE Conference on Dependable Systems and Networks (DSN)*, 2010.

[25] Al-Fares, M. and et. al.., "A Scalable, Commodity Data Center Network Architecture," in *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, 2008.

[26] Amur, H., Cipar, J., Gupta, V., Ganger, G. R., Kozuch, M. A., and Schwan, K., "Robust and Flexible Power-Proportional Storage," in *Proceedings of the 1st ACM symposium on Cloud computing (SoCC)*, 2010.

[27] Anandkumar, A., Bisdikian, C., and Agrawal, D., "Tracking in a Spaghetti Bowl: Monitoring Transactions Using Footprints," in *ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2008.

[28] Ananthanarayanan, G., Agarwal, S., Kandula, S., Greenberg, A., Stoica, I., Harlan, D., and Harris, E., "Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters," in *European Conference on Computer Systems (EuroSys)*, 2011.

[29] Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., and Harris, E., "Reining in the Outliers in Map-Reduce Clusters Using Mantri," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[30] Anderson, E., Hoover, C., Li, X., and Tucek, J., "Efficient Tracing and Performance Analysis for Large Distributed Systems," in *IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2009.

[31] Anderson, E. and Patterson, D., "Extensible, Scalable Monitoring for Clusters of Computers," in *USENIX Large Installation System Administration Conference (LISA)*, 1997.

[32] Apache, "Cloudera flume." `http://archive.cloudera.com/cdh/3/flume/`.

[33] Apache, "Hbase log." `http://hbase.apache.org/book/trouble.log.html`.

[34] Apte, R. and et. al.., "Look Who's Talking: Discovering Dependencies Between Virtual Machines Using CPU Utilization," in *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[35] Arnold, D. C. and Miller, B. P., "Scalable Failure Recovery for High-Performance Data Aggregation," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[36] Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H., "Information and Control in Gray-Box Systems," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[37] Attariyan, M., Chow, M., and Flinn, J., "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[38] Bahl, P., Chandra, R., Greenberg, A. G., Kandula, S., Maltz, D. A., and Zhang, M., "Towards Highly Reliable Enterprise Network Services via Inference of Multi-level Dependencies," in *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, 2007.

[39] Barham, P., Donnelly, A., Isaacs, R., and Mortier, R., "Using Magpie for Request Extraction and Workload Modelling," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[40] Barham, P. and et. al.., "Using magpie for request extraction and workload modelling," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[41] Ben-Yehuda, M., Breitgand, D., Factor, M., Kolodner, H., Kravtsov, V., and Pelleg, D., "NAP: a Building Block for Remediating Performance Bottlenecks via Black Box Network Analysis," in *ACM International Conference on Automatic Computing (ICAC)*, 2009.

[42] Bhatia, S., Kumar, A., Fiuczynski, M. E., and Peterson, L., "Lightweight, High-Resolution Monitoring for Troubleshooting Production Systems," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[43] Bodik, P., Goldszmidt, M., Fox, A., Woodard, D. B., and Andersen, H., "Fingerprinting the Datacenter: Automated Classification of Performance Crises," in *European Conference on Computer Systems (EuroSys)*, 2010.

[44] Botan, I., Cho, Y., Derakhshan, R., Dindar, N., Haas, L., Kim, K., and Tatbul, N., "Federated Stream Processing Support for Real-Time Business Intelligence Applications," in *Proceedings of the VLDB International Workshop on Enabling Real-Time for Business Intelligence (BIRTE)*, 2009.

[45] Brodie, M., Ma, S., Lohman, G., Mignet, L., Modani, N., Wilding, M., Champlin, J., and Sohn, P., "Quickly Finding Known Software Problems via Automated Symptom Matching," in *ACM International Conference on Automatic Computing (ICAC)*, 2005.

[46] Brown, A., Kar, G., and Keller, A., "An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment," in *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2001.

[47] Buzato, L., Vieira, G., and Zwaenepoel, W., "Dynamic Content Web Applications: Crash, Failover, and Recovery Analysis," in *IEEE Conference on Dependable Systems and Networks (DSN)*, 2009.

[48] Buzen, J. P. and Shum, A. W., "MASF – Multivariate Adaptive Statistical Filtering," in *Computer Measurement Group (CMG)*, 1995.

[49] Can, F., "Incremental Clustering for Dynamic Information Processing," *ACM Transactions on Information Systems*.

[50] Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., and Fox, A., "Microreboot: A technique for cheap recovery," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[51] Candea, G., Kiciman, E., Kawamoto, S., and Fox, A., "Autonomous Recovery in Componentized Internet Applications," *Cluster Computing*, 2006.

[52] Cappos, J. and Hartman, J. H., "San Fermin: Aggregating Large Data Sets Using a Binomial Swap Forest," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[53] Cecchet, E. and et. al.., "Performance comparison of middleware architectures for generating dynamic web content," in *ACM/IFIP/USENIX International Conference on Middleware (Middleware)*, 2003.

[54] Chanda, A., Cox, A. L., and Zwaenepoel, W., "Whodunit: Transactional Profiling for Multi-Tier Applications," in *European Conference on Computer Systems (EuroSys)*, 2007.

[55] Chandola, V., Banerjee, A., and Kumar, V., "Anomaly detection: A survey," in *ACM Computing Surveys*, 2009.

[56] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E., "Bigtable: a distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, (Berkeley, CA, USA), pp. 15–15, USENIX Association, 2006.

[57] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E., "Pinpoint: Problem Determination in Large, Dynamic Internet Services," in *IEEE Conference on Dependable Systems and Networks (DSN)*, 2002.

[58] Chen, M. Y., Zheng, A. X., Lloyd, J., Jordan, M. I., and Brewer, E. A., "Failure Diagnosis Using Decision Trees," in *ACM International Conference on Automatic Computing (ICAC)*, 2004.

[59] Chen, X., Zhang, M., Mao, Z. M., and Bahl, P., "Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[60] Cherkasova, L., Ozonat, K. M., Mi, N., Symons, J., and Smirni, E., "Anomaly? Application Change? or Workload Change? Towards Automated Detection of Application Performance Anomaly and Change," in *IEEE Conference on Dependable Systems and Networks (DSN)*, 2008.

[61] Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T., and Fox, A., "Capturing, Indexing, Clustering, and Retrieving System History," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[62] Cohen, I., Goldszmidt, M., Kelly, T., Symons, J., and Chase, J. S., "Correlating Instrumentation Data to System States: a Building Block for Automated Diagnosis and Control," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[63] Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Elmeleegy, K., and Sears, R., "MapReduce Online," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[64] Dai, J., Huang, J., Huang, S., Huang, B., and Liu, Y., "HiTune: Dataflow-based Performance Analysis for Big Data Cloud," in *USENIX Annual Technical Conference (ATC)*, 2011.

[65] De Hoon, M., Imoto, S., Nolan, J., and Miyano, S., "Open Source Clustering Software," *Bioinformatics*, 2004.

[66] Dean, J., "Designs, lessons and advice from building large distributed systems," in *LADIS*, 2009.

[67] Dean, J. and Ghemawat, S., "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, 2008.

[68] Duan, S. and Babu, S., "Guided Problem Diagnosis through Active Learning," in *ACM International Conference on Automatic Computing (ICAC)*, 2008.

[69] Eisenhauer, G. PhD thesis.

[70] Eisenhauer, G., "The evpath library." http://www.cc.gatech.edu/systems/projects/EVPath.

[71] Eisenhauer, G. and Daley, L. K., "Fast Heterogeneous Binary Data Interchange," in *Proceedings of the 9th Heterogeneous Computing Workshop (HCW)*, 2000.

[72] Eisenhauer, G., Schwan, K., and Bustamante, F., "Publish-subscribe for high-performance computing," *IEEE Internet Computing*, vol. 10, pp. 40–47, Jan. 2006.

[73] Erlingsson, U., Peinado, M., Peter, S., and Budiu, M., "Fay: Extensible Distributed Tracing from Kernels to Clusters," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[74] Facebook, "Scribe." `https://github.com/facebook/scribe/wiki`.

[75] Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I., "X-Trace: a Pervasive Network Tracing Framework," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[76] Friedman, N. and et. al.., "Bayesian network classifiers," *Machine Learning*, 1997.

[77] Fu, Q., Lou, J.-G., Wang, Y., and Li, J., "Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis," in *IEEE International Conference on Data Mining (ICDM)*, 2009.

[78] Galstad., E., "Nagios Enterprises LLC.," 2008.

[79] Gärtner, F. C., "Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments," *ACM Computing Surveys*, 1999.

[80] Gavrilovska, A., Schwan, K., and Oleson, V., "A Practical Approach for 'Zero' Downtime in an Operational Information System," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2002.

[81] Gavrilovska, A., Schwan, K., Amur, H., Krishnan, B., Vidyashankar, J., Wang, C., and Wolf, M., "Understanding and Managing IT Power Consumption: A Measurement-Based Approach," in *Energy Effcient Thermal Management of Data Centers, ed. Yogendra Joshi and Pramod Kumar, Springer*, 2012.

[82] Gedik, B., Andrade, H., Wu, K.-L., Yu, P. S., and Doo, M., "SPADE: the System S Declarative Stream Processing Engine," in *ACM Special Interest Group on Management of Data (SIGMOD)*, 2008.

[83] Ghanbari, S., Soundararajan, G., and Amza, C., "A Query Language and Runtime Tool for Evaluating Behavior of Multi-Tier Servers," in *ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2010.

[84] Goldszmidt, M., Budiu, M., Zhang, Y., and Pechuk, M., "Toward Automatic Policy Refinement in Repair Services for Large Distributed Systems," *SIGOPS Operating Systems Review*, 2010.

[85] Gu, W. and et. al.., "Falcon: On-line Monitoring and Steering of Parallel Programs," *Concurrency: Practice and Experience*.

[86] Gu, X., Papadimitriou, S., Yu, P. S., and Chang, S.-P., "Toward Predictive Failure Management for Distributed Stream Processing Systems," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2008.

[87] Gu, X. and Wang, H., "Online Anomaly Prediction for Robust Cluster Systems," in *IEEE International Conference on Data Engineering (ICDE)*, 2009.

[88] Guerraoui, R. and Schiper, A., "Software-Based Replication for Fault Tolerance," *Computer*, 1997.

[89] Gunawi, H. S., Do, T., Joshi, P., Alvaro, P., Hellerstein, J. M., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Sen, K., and Borthakur, D., "FATE and DESTINI: a Framework for Cloud Recovery Testing," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[90] Guo, Z., Zhou, D., Lin, H., Yang, M., Long, F., Deng, C., Liu, C., and Zhou, L., "G2: a Graph Processing System for Diagnosing Distributed Systems," in *USENIX Annual Technical Conference (ATC)*, 2011.

[91] Hewlett-Packard, "Worldcup98 logs." `http://ita.ee.lbl.gov/`.

[92] Hiltunen, M., Schlichting, R., and Ugarte, C., "Building Survivable Services Using Redundancy and Adaptation," *Computers, IEEE Transactions on*, 2003.

[93] Hoogenboom, P. and Lepreau, J., "Computer System Performance Problem Detection Using Time Series Models," in *USENIX Annual Technical Conference (ATC)*, 1993.

[94] Hu, L., Schwan, K., Gulati, A., Zhang, J., and Wang, C., "Net-Cohort: Detecting and Managing VM Ensembles in Virtualized Data Centers," in *ACM International Conference on Automatic Computing (ICAC)*, 2012.

[95] Huebscher, M. C. and McCann, J. A., "A Survey of Autonomic Computing – Degrees, Models, and Applications," *ACM Computing Surveys*, 2008.

[96] Inaba, M. and et. al.., "Applications of Weighted Voronoi Diagrams and Randomization to Variance-based K-Clustering: (extended abstract)," in *Proceedings of the Tenth Annual Symposium on Computational Geometry (SCG)*, 1994.

[97] Isci, C., Wang, C., Bhatt, C., Shanmuganathan, G., and Holler, A., "Process Demand Prediction for Distributed Power and Resource Management," U.S. Patent, 2011.

[98] Jain, N., Mahajan, P., Kit, D., Yalagandula, P., Dahlin, M., and Zhang, Y., "Network Imprecision: a New Consistency Metric for Scalable Monitoring," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[99] Jiang, G., Chen, H., and Yoshihira, K., "Discovering Likely Invariants of Distributed Transaction Systems for Autonomic System Management," *Cluster Computing (Springer)*, 2006.

[100] Jiang, G., Chen, H., Yoshihira, K., and Saxena, A., "Ranking the Importance of Alerts for Problem Determination in Large Computer Systems," in *ACM International Conference on Automatic Computing (ICAC)*, 2009.

[101] Jiang, M., Munawar, M. A., Reidemeister, T., and Ward, P. A., "Automatic Fault Detection and Diagnosis in Complex Software Systems by Information-Theoretic Monitoring," in *IEEE Conference on Dependable Systems and Networks (DSN)*, 2009.

[102] Jiang, M., Munawar, M. A., Reidemeister, T., and Ward, P. A., "System Monitoring with Metric-Correlation Models: Problems and Solutions," in *ACM International Conference on Automatic Computing (ICAC)*, 2009.

[103] Johnson, Jr., A. M. and Malek, M., "Survey of Software Tools for Evaluating Reliability, Availability, and Serviceability," *ACM Computing Surveys*, 1988.

[104] Kandula, S., Mahajan, R., Verkaik, P., Agarwal, S., Padhye, J., and Bahl, P., "Detailed diagnosis in enterprise networks," in *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, 2009.

[105] Kang, H., Chen, H., and Jiang, G., "PeerWatch: a Fault Detection and Diagnosis Tool for Virtualized Consolidation Systems," in *ACM International Conference on Automatic Computing (ICAC)*, 2010.

[106] Kasick, M. P., Tan, J., Gandhi, R., and Narasimhan, P., "Black-box Problem Diagnosis in Parallel File Systems," in *USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[107] Kavulya, S. P., Daniels, S., Joshi, K. R., Hiltunen, M. A., Gandhi, R., and Narasimhan, P., "Draco: Statistical diagnosis of chronic problems in large distributed systems," in *IEEE Conference on Dependable Systems and Networks (DSN)*, 2012.

[108] Kephart, J. O., "Autonomic Computing: the First Decade," in *ACM International Conference on Automatic Computing (ICAC)*, 2011.

[109] Kesavan, M., Gavrilovska, A., and Schwan, K., "Elastic Resource Allocation in Datacenters: Gremlins in the Management Plane," in *VMware Technical Journal*, 2012.

[110] Kiciman, E. and Fox, A., "Detecting Application-Level Failures in Component-based Internet Services," *IEEE Trans. on Neural Networks: Special Issue on Adaptive Learning Systems in Communication Networks*, 2005.

[111] Kiciman, E. e. a., "AjaxScope: a Platform for Remotely Monitoring the Client-Side Behavior of Web 2.0 Applications," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[112] Kim, M., Sumbaly, R., and Shah, S., "Root Cause Detection in a Service-Oriented Architecture," in *ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2013.

[113] Kim, T., Wang, X., Zeldovich, N., and Kaashoek, M. F., "Intrusion Recovery Using Selective Re-Execution," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[114] Ko, S. Y., Yalagandula, P., Gupta, I., Talwar, V., Milojicic, D., and Iyer, S., "Moara: Flexible and Scalable Group-Based Querying System," in *ACM/IFIP/USENIX International Conference on Middleware*, 2008.

[115] Kumar, A. and et. al.., "Data Streaming Algorithms for Efficient and Accurate Estimation of Flow Size Distribution," in *ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2004.

[116] Kumar, S., Talwar, V., Kumar, V., Ranganathan, P., and Schwan, K., "vManage: Loosely Coupled Platform and Virtualization Management in Data Centers," in *ACM International Conference on Automatic Computing (ICAC)*, 2009.

[117] Kumar, V., Schwan, K., Iyer, S., Chen, Y., and Sahai, A., "A State-Space Approach to SLA Based Management," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2008.

[118] Kumar, V., Andrade, H., Gedik, B., and Wu, K.-L., "DEDUCE: at the Intersection of MapReduce and Stream Processing," in *Proceedings of the 13th International Conference on Extending Database Technology (EDBT)*, 2010.

[119] Kumar, V., Cooper, B. F., Eisenhauer, G., and Schwan, K., "iManage: Policy-Driven Self-Management for Enterprise-Scale Systems," in *ACM/IFIP/USENIX International Conference on Middleware*, 2007.

[120] Kutare, M., Eisenhauer, G., Wang, C., Schwan, K., Talwar, V., and Wolf, M., "Monalytics: Online Monitoring and Analytics for Managing Large Scale Data Centers," in *ACM International Conference on Automatic Computing (ICAC)*, 2010.

[121] Lakhina, A., Crovella, M., and Diot, C., "Mining Anomalies Using Traffic Feature Distributions," in *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, 2005.

[122] Lakhina, A. and et. al., "Diagnosing Network-Wide Traffic Anomalies," in *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, 2004.

[123] Lakshminarayan, C., Viswanathan, K., Wang, C., and Talwar, V., "Statistically-based Anomaly Detection in Utility Clouds," U.S. Patent Pending, 2011.

[124] Lee, M., Krishnakumar, A. S., Krishnan, P., Singh, N., and Yajnik, S., "Supporting Soft Real-time Tasks in the Xen Hypervisor," in *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE)*, 2010.

[125] Leners, J. B., Wu, H., Hung, W.-L., Aguilera, M. K., and Walfish, M., "Detecting Failures in Distributed Systems with the Falcon Spy Network," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[126] Leverich, J. and et. al.., "Evaluating Impact of Manageability Features on Device Performance," CNSM '10.

[127] LinkedIn, "Kafka." `http://sna-projects.com/kafka/design.php`.

[128] Liu, X., Guo, Z., Wang, X., Chen, F., Lian, X., Tang, J., Wu, M., Kaashoek, M. F., and Zhang, Z., " D^{3}S: Debugging Deployed Distributed Systems," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[129] Lofstead, J., Zheng, F., Liu, Q., Klasky, S., Oldfield, R., Kordenbrock, T., Schwan, K., and Wolf, M., "Managing Variability in the IO Performance of Petascale Storage Systems," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[130] Logothetis, D. and Yocum, K., "Wide-scale Data Stream Management," in *USENIX Annual Technical Conference (ATC)*, 2008.

[131] Long, D., Muir, A., and Golding, R., "A Longitudinal Survey of Internet Host Reliability," in *Proceedings of the 14th Symposium on Reliable Distributed Systems*, 1995.

[132] Lou, J., Fu, Q., Yang, S., Xu, Y., and Li, J., "Mining Invariants from Console Logs for System Problem Detection," in *USENIX Annual Technical Conference (ATC)*, 2010.

[133] Ma, T., Hillston, J., and Anderson, S., "On the Quality of Service of Crash-Recovery Failure Detectors," in *IEEE Conference on Dependable Systems and Networks (DSN)*, 2007.

[134] Madden, S. R. and et. al., "TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[135] Mansour, M. and Schwan, K., "I-RMI: Performance Isolation in Information Flow Applications," in *ACM/IFIP/USENIX International Conference on Middleware (Middleware)*, 2005.

[136] Marwah, M., Mishra, S., and Fetzer, C., "Enhanced Server Fault-Tolerance for Improved User Experience," in *IEEE Conference on Dependable Systems and Networks (DSN)*, 2008.

[137] Marz, N., "Twitter's storm." `https://github.com/nathanmarz/storm`.

[138] Massie, M. L., Chun, B. N., and Culler, D. E., "The Ganglia Distributed Monitoring System: Design, Implementation And Experience," *Parallel Computing*, 2003.

[139] McDowell, C. E. and Helmbold, D. P., "Debugging Concurrent Programs," *ACM Computing Surveys*, 1989.

[140] Mendes, C. L. and Reed, D. A., "Monitoring Large Systems via Statistical Sampling," *International Journal of High Performance Computing Applications*, 2004.

[141] Meng, S. PhD thesis.

[142] Meng, S., Kashyap, S., Venkatramani, C., and Liu, L., "REMO: Resource-Aware Application State Monitoring for Large-Scale Distributed Systems," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2009.

[143] Meng, S., Wang, T., and Liu, L., "Monitoring Continuous State Violation in Datacenters: Exploring the Time Dimension," in *IEEE International Conference on Data Engineering (ICDE)*, 2010.

[144] Microsystems, S., "XDR: External Data Representation standard." RFC 1014, June 1987.

[145] Murtagh, F., "Complexities of hierarchic clustering algorithms: the state of the art," *Computational Statistics Quarterly*, 1984.

[146] Narayanan, D. and et. al.., "Delay Aware Querying with Seaweed," *The VLDB Journal*.

[147] Neumeyer, L., Robbins, B., Nair, A., and Kesari, A., "S4: Distributed Stream Computing Platform," in *2010 IEEE International Conference on Data Mining Workshops (ICDMW)*, 2010.

[148] Nugent, J. A., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Controlling Your PLACE in the File System with Gray-box Techniques," in *USENIX Annual Technical Conference (ATC)*, 2003.

[149] Oliveira, F., Tjang, A., Bianchini, R., Martin, R. P., and Nguyen, T. D., "Barricade: Defending Systems Against Operator Mistakes," in *European Conference on Computer Systems (EuroSys)*, 2010.

[150] Oppenheimer, D., Ganapathi, A., and Patterson, D. A., "Why Do Internet Services Fail, and What Can Be Done About It?," in *USENIX Symposium on Internet Technologies (USITS)*, 2003.

[151] Ozonat, K., "An Information-Theoretic Approach to Detecting Performance Anomalies and Changes for Large-Scale Distributed Web Services," in *IEEE Conference on Dependable Systems and Networks (DSN)*, 2008.

[152] Pan, X., Tan, J., Kavulya, S., R., G., and Narasimhan, P., "Ganesha: Black-Box Diagnosis of MapReduce Systems," in *Workshop on Hot Topics in Measurement and Modeling of Computer Systems (HotMetrics)*, 2009.

[153] Park, K. and Pai, V. S., "CoMon: a Mostly-Scalable Monitoring System for PlanetLab," *SIGOPS Operating Systems Review*, 2006.

[154] Pertet, S. and Narasimhan, P., "Causes of Failure in Web Applications," tech. rep., CMU-PDL-05-109, 2005.

[155] Rabkin, A. and Katz, R., "Chukwa: a System for Reliable Large-Scale Log Collection," in *USENIX Large Installation System Administration Conference (LISA)*, 2010.

[156] Ren, G., Tune, E., Moseley, T., Shi, Y., Rus, S., and Hundt, R., "Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers," *Micro, IEEE*, 2010.

[157] Repantis, T., Cohen, J., Smith, S., and Wein, J., "Scaling a Monitoring Infrastructure for the Akamai Network," *SIGOPS Operating Systems Review*, 2010.

[158] Reynolds, P., Killian, C., Wiener, J. L., Mogul, J. C., Shah, M. A., and Vahdat, A., "Pip: Detecting the Unexpected in Distributed Systems," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[159] Rijsbergen, C. J. v., *Information Retrieval*. 1979.

[160] Rish, I., Brodie, M., Odintsova, N., Ma, S., and Grabarnik, G., "Real-Time Problem Determination in Distributed Systems Using Active Probing," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2004.

[161] Sambasivan, R. R., Zheng, A. X., De Rosa, M., Krevat, E., Whitman, S., Stroucken, M., Wang, W., Xu, L., and Ganger, G. R., "Diagnosing Performance Changes by Comparing Request Flows," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[162] Schwefel, H. P. and Antonios, I., "Performability Models for Multi-Server Systems with High-Variance Repair Durations," in *IEEE Conference on Dependable Systems and Networks (DSN)*, 2007.

[163] Shannon, C. E., "A Mathematical Theory of Communication," *Bell System Technical Journal*, 1948.

[164] Shen, K., Stewart, C., Li, C., and Li, X., "Reference-Driven Performance Anomaly Identification," in *ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2009.

[165] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C., "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure," tech. rep., Google, 2010.

[166] Sottile, M. and et. al.., "Supermon: a High-speed Cluster Monitoring System," Cluster Computing, 2002.

[167] Soundararajan, V. and et. al.., "The Impact of Management Operations on the Virtualized Datacenter," in *International Symposium on Computer Architecture (ISCA)*, 2010.

[168] Stewart, C. and Shen, K., "Performance Modeling and System Management for Multi-component Online Services," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[169] Tak, B. C., Tang, C., Zhang, C., Govindan, S., Urgaonkar, B., and Chang, R. N., "vpath: precise discovery of request processing paths from black-box observations of thread and network activities," in *USENIX Annual Technical Conference (ATC)*, 2009.

[170] Tan, J., Pan, X., Marinelli, E., Kavulya, S., Gandhi, R., and Narasimhan, P., "Kahuna: Problem Diagnosis for Mapreduce-based Cloud Computing Environments," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2010.

[171] Tan, Y. and Gu, X., "On Predictability of System Anomalies in Real World," in *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2010.

[172] Tan, Y., Gu, X., and Wang, H., "Adaptive System Anomaly Prediction for Large-Scale Hosting Infrastructures," in *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, 2010.

[173] Tan, Y., Nguyen, H., Shen, Z., Gu, X., Venkatramani, C., and Rajan, D., "PREPARE: Predictive Performance Anomaly Prevention for Virtualized Cloud Systems," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2012.

[174] Thereska, E., Salmon, B., Strunk, J., Wachs, M., Abd-El-Malek, M., Lopez, J., and Ganger, G. R., "Stardust: Tracking Activity in a Distributed Storage System," *SIGMETRICS Performance Evaluation Review*, 2006.

[175] Tibco Software Inc., *Tibco ActiveEnterprise: XML Tools*. http://www.tibco.com/solutions/products/extensibility/.

[176] Traeger, A., Deras, I., and Zadok, E., "DARC: Dynamic Analysis of Root Causes of Latency Distributions," in *ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2008.

[177] Tucek, J., Lu, S., Huang, C., Xanthos, S., and Zhou, Y., "Triage: Diagnosing Production Run Failures at the User's Site," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[178] Vaidyanathan, K., Harper, R. E., Hunter, S. W., and Trivedi, K. S., "Analysis and implementation of software rejuvenation in cluster systems," in *ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2001.

[179] Van Renesse, R., Birman, K. P., and Vogels, W., "Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining," *ACM Transactions on Computer Systems*, 2003.

[180] Viswanathan, K., Choudur, L., Talwar, V., Wang, C., MacDonald, G., and Satterfield, W., "Ranking Anomalies in Data Centers," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2012.

[181] Viswanathan, K., Lakshminarayan, C., Satterfield, W. J., Talwar, V., and Wang, C., "System and Method for Ranking Anomalies," U.S. Patent Pending, 2013.

[182] Voldemort, "Project voldemort - a distributed database." `http://project-voldemort.com/`.

[183] Wang, C., "EbAT: Online Methods for Detecting Utility Cloud Anomalies," in *the 6th Middleware Doctoral Symposium (MDS 2009) in conjunction with Middleware 2009*, 2009.

[184] Wang, C., Kavulya, S. P., Tan, J., Hu, L., Kutare, M., Kasick, M., Schwan, K., Narasimhan, P., and Gandhi, R., "Performance Troubleshooting in Data Centers: An Annotated Bibliography," in *ACM SIGOPS Operating Systems Review*, 2013.

[185] Wang, C., Rayan, I. A., Eisenhauer, G., Schwan, K., Talwar, V., Wolf, M., and Huneycutt, C., "VScope: Middleware for Troubleshooting Time-Sensitive Data Center Applications," in *ACM/IFIP/USENIX International Conference on Middleware (Middleware)*, 2012.

[186] Wang, C., Rayan, I. A., and Schwan, K., "Faster, Larger, Easier: Reining Real-Time Big Data Processing in Cloud," in *ACM/IFIP/USENIX International Conference on Middleware (Middleware)*, 2012.

[187] Wang, C., Schwan, K., Talwar, V., Eisenhauer, G., Hu, L., and Wolf, M., "A Flexible Architecture Integrating Monitoring and Analytics for Managing Large-Scale Data Centers," in *ACM International Conference on Automatic Computing (ICAC)*, 2011.

[188] Wang, C., Talwar, V., Schwan, K., and Ranganathan, P., "Online Detection of Utility Cloud Anomalies Using Metric Distributions," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2010.

[189] Wang, C., Talwar, V., and Ranganathan, P., "Cloud Anomaly Detection Using Normalization, Binning and Entropy Determination." U.S. Patent Pending, 2010.

[190] Wang, C., Talwar, V., and Ranganathan, P., "Network System Management," U.S. Patent Pending, 2010.

[191] Wang, C., Viswanathan, K., Choudur, L., Talwar, V., Satterfield, W., and Schwan, K., "Statistical Techniques for Online Anomaly Detection in Data Centers," in *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2011.

[192] Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M. I., "Detecting Large-Scale System Problems by Mining Console Logs," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[193] Yalagandula, P. and Dahlin, M., "A Scalable Distributed Information Management System," in *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, 2004.

[194] Yuan, C., Lao, N., Wen, J.-R., Li, J., Zhang, Z., Wang, Y.-M., and Ma, W.-Y., "Automated Known Problem Diagnosis with Event Traces," in *European Conference on Computer Systems (EuroSys)*, 2006.

[195] Zhang, Z., Zhan, J., Li, Y., Wang, L., Meng, D., and Sang, B., "Precise Request Tracing and Performance Debugging for Multi-Tier Services of Black Boxes," in *IEEE Conference on Dependable Systems and Networks (DSN)*, 2009.

[196] Zhao, Y., Tan, Y., Gong, Z., Gu, X., and Wamboldt, M., "Self-Correlating Predictive Information Tracking for Large-Scale Production Systems," in *ACM International Conference on Automatic Computing (ICAC)*, 2009.

[197] Zheng, F., Abbasi, M., Docan, C., Lofstead, J., Liu, Q., Klasky, S., Prashar, M., Podhorszki, N., Schwan, K., and Wolf, M., "PreDatA - Preparatory Data Analytics on Peta-Scale Machines," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[198] Zheng, F., Yu, H., Hantas, C., Wolf, M., Eisenhauer, G., Schwan, K., Abbasi, H., and Klasky, S., "GoldRush: Resource Efficient In Situ Scientific Data Analytics Using Fine-Grained Interference Aware Execution," in *ACM/IEEE International Conference for High Performnance Computing, Networking, Storage and Analysis (SC)*, 2013.

[199] Zheng, Z. and Lyu, M., "A QoS-Aware Fault Tolerant Middleware for Dependable Service Composition," in *IEEE Conference on Dependable Systems and Networks (DSN)*, 2009.

[200] Zhou, J., Zhang, C., Tang, H., Wu, J., and Yang, T., "Programming Support and Adaptive Checkpointing for High-Throughput Data Services with Log-Based Recovery," in *IEEE Conference on Dependable Systems and Networks (DSN)*, 2010.

[201] Zhu, Q. and Yuan, C., "A Reinforcement Learning Approach to Automatic Error Recovery," in *IEEE Conference on Dependable Systems and Networks (DSN)*, 2007.

VITA

Chengwei Wang is a Ph.D. candidate of Computer Science in College of Computing at Georgia Institute of Technology. His Ph.D. thesis focuses on investigating novel system and algorithms for troubleshooting large scale data center and its applications. He has spent summers as research/engineering intern in Amazon, HP Labs and VMware. Before pursuing Ph.D. in U.S., he was a R&D Engineer in IBM China Research Laboratory (IBMCRL). He graduated with a Master degree and a Bachelor degree in Computer Science from Huazhong University of Science and Technology in 2003 and 2006, respectively.