Monalytics: Online Monitoring and Analytics for Managing Large Scale Data Centers

Mahendra Kutare
College of Computing
Georgia Institute of Technology
Atlanta, GA 30318, USA
imax@cc.gatech.edu

Greg Eisenhauer
College of Computing
Georgia Institute of Technology
Atlanta, GA 30318, USA
eisen@cc.gatech.edu

Chengwei Wang
College of Computing
Georgia Institute of Technology
Atlanta, GA 30318, USA
wangcw@cc.gatech.edu

Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA 30318, USA
schwan@cc.gatech.edu

Vanish Talwar
HP Labs
1501 Page Mill Road
Palo Alto, CA 94304 USA
vanish.talwar@hp.com

Matthew Wolf
College of Computing
Georgia Institute of Technology
Atlanta, GA 30318, USA
mwolf@cc.gatech.edu

ABSTRACT

To effectively manage large-scale data centers and utility clouds, operators must understand current system and application behaviors. This requires continuous monitoring along with online analysis of the data captured by the monitoring system. As a result, there is a need to move to systems in which both tasks can be performed in an integrated fashion, thereby better able to drive online system management. Coining the term 'monalytics' to refer to the combined monitoring and analysis systems used for managing large-scale data center systems, this paper articulates principles for monalytics systems, describes software approaches for implementing them, and provides experimental evaluations justifying principles and implementation approach. Specific technical contributions include consideration of scalability across both 'space' and 'time', the ability to dynamically deploy and adjust monalytics functionality at multiple levels of abstraction in target systems, and the capability to operate across the range of application to hypervisor layers present in large-scale data center or cloud computing systems. Our monalytics implementation targets virtualized systems and cloud infrastructures, via the integration of its functionality into the Xen hypervisor.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; D.4.4 [**Communication Management**]: Network Communications

ICAC'10, June 7–11, 2010, Washington, DC, USA.

General Terms

Design, Measurement

Keywords

Monitoring, Management

1. INTRODUCTION

Performance and program monitoring are well-established areas of research. Traditionally concerned with performance debugging [23], a more recent focus of online monitoring has been to help deal with the ever-increasing complexity and scale of modern IT and data center facilities. Here, monitoring is used to continually measure and assess current system or application behaviors [18], to detect and diagnose problems, or even for purposes of business analytics. The data used by such analyses is captured by a wide variety of tools, operating in specific subsystems, running at different levels of abstraction [14], on entire parallel machines [22, 23], or addressing the distributed nature of underlying infrastructures [19, 21, 26]. These data collection and dissemination tools may be integrated with problem diagnosis systems [20], with program development systems (to better understand the applications being monitored) [22], and they may themselves be managed to improve how monitoring is performed [16]. In most such settings, however, the primary purpose of monitoring is to improve the ways in which systems and applications operate, measured in terms of performance, reliability, power usage, the ability to meet service level agreements (SLAs), and similar metrics important to applications and IT infrastructure providers [17]

Monitoring to Manage Large-scale Systems..

Our research is developing methods and infrastructures to improve the manageability of future data center systems [17]. This paper is focused on a necessary element of system management, which is efficient and scalable online system monitoring. Concerning scale, recent reports indicate that already, up to 25% of enterprise data today is from systems monitoring, with almost 240 terabytes produced annually,

and this number is only going to increase with next generation facilities. Furthermore, scale goes beyond raw system size in that one also has to take into account the multiple time and length scales at which different system components and levels of abstraction operate. Concerning the 'length' scale, consider operator queries about current data center health for large-scale systems vs. providing detailed information about the state of a specific disk subsystem, for instance. Concerning time scales, consider the high rate at which web requests are received and serviced by system-level threads running across multiple processors, compared with the lower rate at which the virtual machines running these threads migrate across machines when being consolidated; or consider the management done to differentiate service levels for (high rate) disk requests vs. the relatively low-rate power management actions applied to the processors that run such disk-centric applications

The examples described above illustrate that when the purpose of monitoring is to better manage systems or applications, monitoring must operate across multiple time scales, across different size systems, and at multiple levels of abstraction – from application-centric entities like 'requests' to infrastructure-centric entities like blades or racks. Furthermore, monitoring must go beyond data capture and dissemination to also understanding and analyzing captured data [16], in addition to support intelligent problem determination methods [11], as needed by subsequent management actions.

Monalytics..

We refer to our combined monitoring and analysis system as 'monalytics'. This paper identifies and explores several important properties of 'monalytics' that are key to its applicability to large-scale IT infrastructures.

Data-local analysis – for low latency response, that is, in order to limit the delays between when monitoring data is first captured to when interesting insights are derived from that data, it must be possible to perform select analyses 'close' to data sources. While traditional systems have eschewed such solutions due to the potential perturbation caused by additional monitoring loads [23], given that modern machines are increasingly bound in performance by memory bandwidth rather than CPU speed, we advocate an approach in which data-near analysis is used both to reduce monitoring data volume and to rapidly gain insights from captured data. In other words, we posit that it is often cheaper to quickly analyze data and then send out data summaries or abstracts than it is to copy raw captured data items – this fact has also been shown to hold for data dissemination across networked machines [7], and we note that for the same reasons, physical systems using smart sensors are increasingly common. For monitoring, this implies the need to combine the flexible data capture done in traditional monitoring systems [23] with low overhead methods for associating light-weight analysis methods with capture mechanisms.

Operation at multiple time scales – it is well-known that monitoring rates (e.g., sampling rates) depend on the artifacts and behaviors being watched, as do window sizes (e.g., sample sizes). Further, systems exhibit multi-time scale behaviors for different hardware sensors (e.g., slowly changing thermal sensors vs. rapidly changing cache miss rates) as well as for instrumented software. As a result, adjustable monitoring rates, window sizes, and associated noise reduction and data filtering constitute the base functionality required by any monalytics system. A corollary is the need to analyze data across multiple time scales, which for analytics, typically requires analysis-specific methods, as when trying to correlate observed temperature changes with changes in IT loads and/or processor utilization. Our monalytics infrastructure makes it easy to adjust rates and window sizes, and to associate filtering and data smoothing codes with capture mechanisms. Also shown in this paper is a useful multi-scale analysis technique, using entropy-based methods.

Scalability to different system sizes – as with the 'scope in time' implied by window sizes, monitoring must also use 'scope in space', meaning that it must be possible to limit the number of entities being monitored to those of current interest. A corollary is that for different entities of interest, it must be possible to use different ways of attaining scale, an example being the use of hierarchical monitoring structures like aggregation trees [28] for physical entities such as processors, blades, and racks vs. the use of peer-to-peer relationships for information dissemination and aggregation in distributed systems [21]. To permit such variety, we use *zones* as useful partitions of physical systems. Within each zone, data capture agents are connected to monitoring brokers using zone-specific structures, where the term ***monitoring topology*** refers to the structure used to connect data capture/analysis agents with data aggregators/monitoring brokers. Within the agents and brokers providing the physical containers for capturing and analyzing monitoring data, each specific monitoring task being performed is represented as a computational communication graph [3], and agents and brokers are internally multiplexed to operate any number of those graphs.

Runtime discovery, configuration, and adaptation – data center machines are subject to dynamic change in usage and load, but at the same time, we desire predictable latencies from monalytics. This means that it must be possible to change what, where, and how monitoring is done, including to deploy at runtime monitoring and analysis codes to where they are needed, to change the actual monalytics methods being used as well as the associated monalytics structures, and to dynamically discover and attach monalytics to new data sources when needed. Our work uses dynamic binary code generation and deployment to help obtain these capablities.

In summary, scalability demands that the monalytics components for data collection, aggregation, and analysis be flexible, ranging from simple centralized solutions to highly distributed ones. Further, solutions should scale up and down efficiently, particularly when monalytics drives real-time decision making like resource management to guarantee application SLAs and/or meet data center-level requirements like constraints on power usage [17].

Contributions and Results..

This paper makes the following technical contributions:

- The concept of monalytics is introduced, along with a software architecture for realizing it in large-scale data center systems.
- Scalability is also sought with respect to time and size scales, and to operate across the different levels of abstraction at which modern IT systems are described and implemented. The time scale is particularly relevant

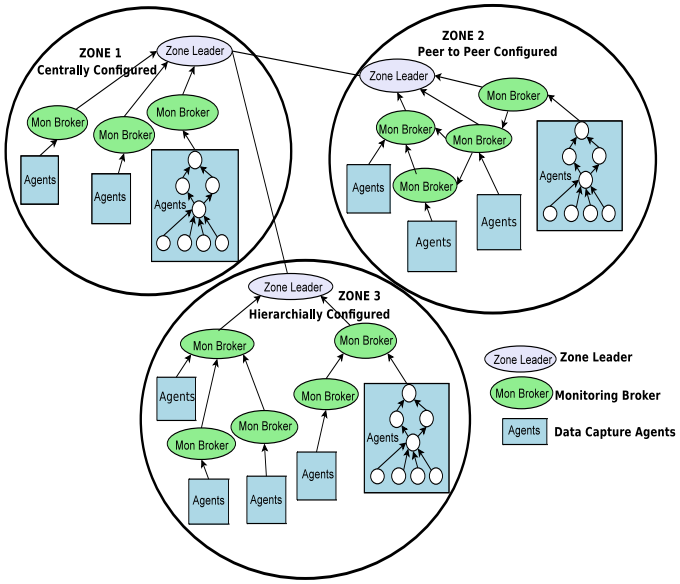

Figure 1: Monalytics Topology

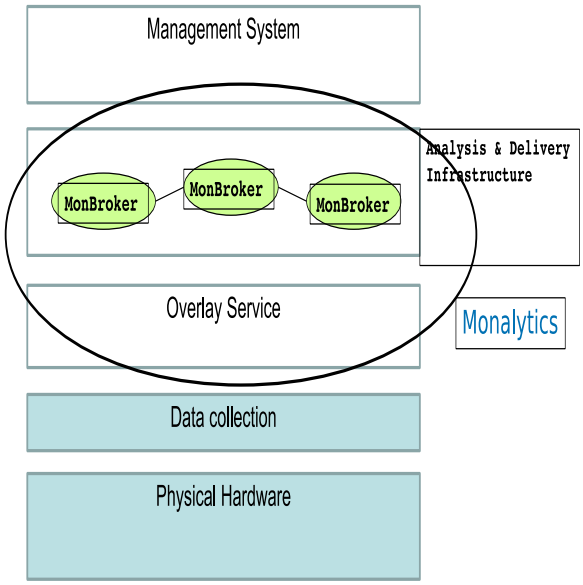

Figure 2: Monalytics Logical View

to online management, since it may be important to quickly react to dynamic changes in conditions and requirements.

- The utility of monalytics and its scalability principles are demonstrated with representative hardware and applications, and with micro-benchmarks.

Monalytics has been implemented for virtualized computing infrastructures. A prototype constructed for the Xen open source hypervisor is shown to add little to no additional overheads to the execution of typical data center codes. A transactional web application, variable request loads, and performance and fault behaviors are recognized and controlled, and the potential for scalability is demonstrated via adaptable filtering and 'failure-proportional' monalytics actions.

2. SYSTEM ARCHITECTURE AND IMPLEMENTATION

The realization of monalytics is based on three principles: (1) monalytics actions are deployed as a computational communication graph in the data center; (2) such graphs are elastic and reconfigurable based on monalytics needs, current state, and load in the data center; and (3) graph layouts and implementations are dynamic with respect to their use of centralized, hierarchical, or peer-to-peer structures, leveraging the best of the infrastructure but without having to subscribe to any one of them statically.

These principles give rise to the software architecture described in Fig. 1. *Agents* capture and locally process desired data; they reside at multiple levels of abstraction in target systems, including at application-, system-, and hypervisor-level. Agents also access hardware and physical sensors, such as hardware counters provided by computing platforms and power draw values exported by PDUs. *Brokers* aggregate and analyze outputs from multiple agents, and they are linked in ways that respect available communication and hardware structures. Each set of agents/broker are internally multiplexed to execute multiple logical structures – monalytics actions represented as computational communication graphs – that each represent specific captured data and the analysis methods applied to it. In other words, as with multiple threads in a single process, agents and brokers internally maintain and operate multiple monalytics graphs. Agent representations differ depending on target systems being monitored, including the use of specialized device drivers when interfacing agents with hardware data collection systems. Brokers can execute in specialized virtual machines (e.g., management VMs), on dedicated hosts (e.g., manageability engines), or both. Important in this context is that it must be possible to separately provision brokers and agents, so that latency and QoS guarantees can be made for monitoring and management, unaffected by current application actions and loads. Earlier work described how agents and brokers interact in a uniform manner, using a channel abstraction, termed m(anagement)channels [17]. The current implementation of monalytics has not yet been integrated with m-channels, but that work is in progress. *Zones* indicate physical subsystems, such as sets of racks on a single network switch in a data center, front end vs. back end machines, etc. Applications, typically comprised of *ensembles* of virtual machines, can span multiple zones, an example being web applications whose request schedulers run on data center front end machines, whereas its database servers run on the center's backend machines in a different zone. Zones, therefore, are a vehicle for delineating data center subsystems or substructures, or even the multiple physical data centers located in a single public/private cloud.

Applying these principles, Fig. 2 shows the logical view, with our core contribution being an overlay service and a collection of composed monitoring brokers that perform correlation, aggregation, and analysis functions. The overlay service provides the underlying communication/routing mechanisms as well as discovery and namespace registration for a collection of monitoring brokers. Together, they result in a monitoring computation graph (see Fig. 1), with multiple graphs multiplexed onto shared brokers, as also indicated.

The computational communication graphs used for monalytics perform the distributed monitoring functions in the data center in a cooperative manner. They are created and operate as described next.

The bootstrap creation phase starts by an initial assignment of monitoring brokers to logical zones by a planning algorithm at a central management station (CMS) [18]. Physically, a zone is a collection of nodes associated with a unique identifier communicated to the brokers on each physical node. After the assignment to zones, the monitoring brokers elect a leader for their respective zones using a leader election algorithm. The elected leaders of the respective zones then elect a leader among themselves (leader of leaders). The resulting inter-connection among the leaf node monitoring brokers, zone leaders, and inter-zone leaders is the initial monalytics graph after bootstrap. However, brokers only provide the physical containers that run monalytics graphs. This means that a primary role of leaders is to deploy and configure monalytics graphs across sets of brokers and supervise their execution.

During system operation, the role of monitoring brokers is to execute two functions: (1) they run the portions of monalytics graphs assigned to them, applying data aggregation and analysis functions on monitoring data streams, and raising alerts of anomalous behavior when detected (e.g., to enable local policies to be applied 'close' to the source of monitoring data); and (2) they propagate raw and/or analyzed local data streams to the other entities participating in each of their computation graphs, often ending at the zone-leader for zone-level aggregation. Along with such 'data plane' operations, there are 'control' actions, an example being a reaction to the fact that zone-level aggregation completion time exceeds an acceptable threshold. When that happens, the zone-leader may trigger reconfiguration of the monalytics graph within the zone, including those that switch the interconnections between monitoring brokers and leader from say, a centralized configuration to a multi-hierarchy topology, a completely peer-to-peer topology, or a combination of both.

When the aggregation for a polling interval completes, this triggers the execution of some management policy on the aggregated data. Processes like these occur across all levels of the monalytics graphs and may continue across zones, except that it now takes place among zone-level leaders. Similar reconfiguration of the computation graph (as within a zone) can take place across the leaders of the zones based on system load.

In summary, the monalytics system design leads to a proactive, elastic, and distributed monitoring system that supports hybrid topologies in a flexible manner. Its desirable properties include (1) applying analysis/anomaly detection policies local/close to source of monitoring data, (2) satisfying SLAs on aggregation/analysis completion times, such as an upper bound on those times, irrespective of scale in the system, (3) incurring overhead proportional and better than the scale and load in the system achieving elastic balancing, (4) flexibility of configuration and definition of SLA/metrics.

Fig. 3 depicts the Xen prototype of the monalytics infrastructure. The monitoring agents provide mechanisms to start, stop, and pause monitoring tasks. Each such task is represented as a computational communication graph, with graph nodes representing simple data processing operations and links denoting data transfers. Graphs can operate

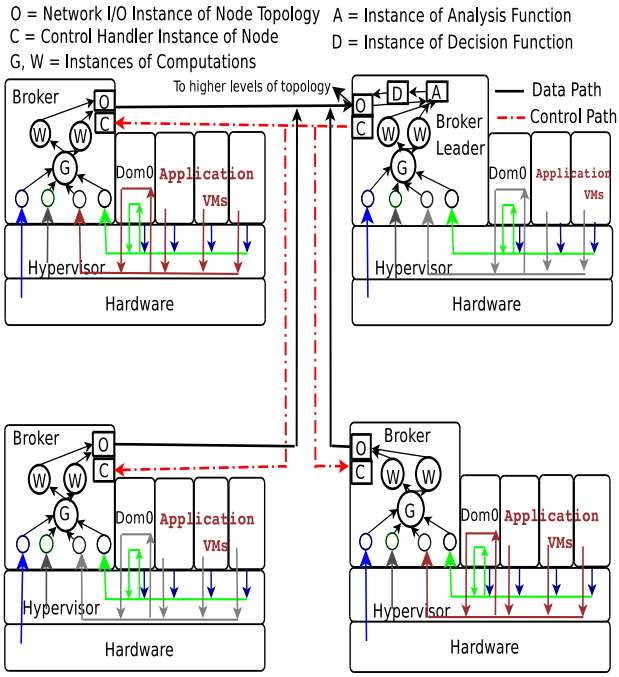

Figure 3: A Simplified Monalytics Architecture

within individual entities, such as an agent implementing a local feedback loop, and they can extend across multiple agents, brokers, and ultimately, zones.

The monalytics implementation separates control and data planes, where control actions can dynamically change the ways in which data is captured, processed, and transported. Current agents, for instance, are flexible with respect to how much and at what rates data is captured, and as to how data is combined and filtered. Most analyses are performed in brokers, including data aggregation from multiple agents. The current implementation maintains a list of predefined functions for these purposes, whereas runtime binary code generation and parametric controls are used to affect how agents operate. Selected brokers – leaders – run the 'coordinator' functions that control the data plane, i.e., create and re-configure the computational communication graphs that implement monalytics functionality.

We next describe in more detail the data capture agents and the analysis functions used in agents and brokers.

Data Measurement, Capture, and Analysis

The monalytics infrastructure offers several abstractions for capturing and analysis data:

1. ***Data capture*** – underlying monalytics are subsystem- and tool-specific components used for extracting data from systems and applications; to retain information about how this is done, monalytics maintains for each captured data item the tuple $\langle capture\_id, level\_id, sampling\_rate, sensor\_info \rangle$.
   capture_id represents the mechanism with which data is acquired, such as the unix */proc* or Xen's libvirt. *level_id* identifies the level of abstraction at which data is captured, such as the virtualization layer, operating system, platform or rack, etc. The actual data acquired in this fashion, sensor_info, is described as

$\langle time\_stamp, name, value \rangle$ tuple of metric data. The time_stamp represents the time at which metric is captured, and *value* contains the metric's value. $sampling\_rate$ represents the frequency of sampling monitoring metrics.

2. ***Data representation*** – once captured, data is represented as groups or windows. ***group*** – a group of data is always of some specific type and is captured in some context $group\_type, sensor\_info$, where measurement are collected in sets like cpu metrics, disk metrics, cpu and memory metrics e.t.c. and $sensor\_info$ is as above; an example is a group of cpu metrics that describe cpu utilization as $cpu\_system, cpu\_user, and\ cpu\_idle$ values; ***window*** – most analysis methods operate over window of values, motivating the implementation of a window as $\langle window\_type, window\_size, sensor\_info \rangle$ tuple; $window\_type$ can be of type time or event, $window\_size$ represents sizes such as 10sec or events, and $sensor\_info$ are as above.

To combine monitoring with analysis, monalytics permits low-level analysis actions to be associated with data capture. Such actions are represented as binary codes and can be customized to each specific data capture activity. Actions associated with data capture, for example, include a sample of violation counts for a group of metrics used – termed concise sample. Concise sample analysis is applied as post-processing actions to the event or time window actions to create a violation event for the last few seconds or events. It is likely that additional analysis actions will become built-in elements of data capture activities as the monalytics system evolves. Higher level analyses, however, are use- and application-specific; they are programmed explicitly and/or use mathematical and statistics libraries, with future work to offer interfaces to mathematical libraries like MatLab to assist end users in programming complex analysis actions.

Data Transport and Processing

The data transport and processing layer of monalytics uses the EVPath eventing system [23] to implement the computational communication graphs embedded in agents and brokers. Graphs are constructed as sets of linked *stones* traversed by *events* (i.e., structured data objects), where stones can perform event data filtering, data transformation, event multiplexing and demultiplexing, and event transmission to other stones. Stones can be linked within or across address spaces, the former via shared memory, the latter via standard communication protocols. Events are represented in efficient binary forms, via a portable binary format implemented in monalytics. Data is converted into this format upon entry to the monalytics system (e.g., via XML-to-binary conversion). Each stone is associated with a set of *actions*. Actions are codes that operate on the messages/events handled by the stones. Stones also have the ability to temporarily hold events, retain limited state, and can be created or destroyed at runtime. Finally, higher level actions defined on sets of stones can reconfigure them and their linkages [18].

Stones can carry out arbitrary actions. (a) An *output action* causes a stone to send messages to a target stone across a network link. (b) A *terminal action* specifies an application handler that will consume incoming events. (c) A *filter action* allows handlers that filter incoming data to determine if it will be passed to subsequent stones. (d) A *split action* allows incoming events to be sent to multiple stones. The target stones of a split action can be dynamically changed by adding/removing stones from the split target on the fly. (e) A *transform action* transforms event data from one data type to another, and it can be used to perform complex calculations on events, such as sampling, averaging, and compression. Actions are implemented by handlers written in C or via runtime generated binary codes, the latter specified in E-Code, a portable subset of the C language. Runtime action deployment, then, uses dynamic linking or dynamic code generation, respectively.

Monalytics implementations can be built on substrates other than EVPath, of course, but there are multiple principles embedded in EVPath that contribute to the scalable design of monalytics. These include the runtime creation and deployment of meaningful data capture and analysis actions, the ability to dynamically reconfigure monalytics (both in terms of actions and graph structure), the capability to operate efficiently both 'in the small', e.g., in a single address space, and 'in the large', i.e., across the many agents and brokers present in a large scale data center. Efficiency 'in the small' is provided in part by use of compact binary representations of data coupled with compiled action implementations. The compact binary formats in current use include formats that define groups, event and time windows, encodings for various metrics, etc. Efficiency 'in the large' is supported by runtime reconfiguration of the monalytics structures spanning many agents and brokers.

The default EVPath execution model is push-based, where agents continuously capture sensor data, process it via local actions, and send it to brokers (as defined by monitoring graphs), and brokers carry out global actions. To support adhoc monalytics queries, we have added a pull-based model, where queries posed to brokers cause agents to acquire and provide certain data, as and when needed. In addition, we are currently implementing efficient techniques for storing or caching select and/or adhoc state present in monalytics structures, a concrete example being a DHT used to maintain state for popular or common queries [26]. The methods available for controlling how monalytics graphs operate are described next.

Controlling Monalytics

Monalytics is implemented to separate control from data paths, where data capture, processing, and forwarding are done via monalytics structures, and control actions are taken by coordinators associated with these structures. Technically, this means that agents and brokers jointly execute monalytics graphs, but that select brokers like zone leaders also run coordinator processes that control how monalytics graphs function.

Control actions of interest to this paper include the ability to dynamically change (a) the capture rate for monitoring metrics data, (b) the monitoring graphs' processing points, i.e., actions, (c) the targets for the processing points in the monitoring graphs and between agents and brokers, the latter enabling higher level functions that reconfigure graph topologies. For (c), for instance, a *split* stone can use a 'target list' of various targets to be used, and this list can be changed at runtime. Changes may be triggered by stone arrival/departure or by link creation/deletion. The experimental evaluations in Section 4 use (a) and (b), for instance, when dynamically switching from lightweight monitoring for

abnormal behavior detection to undertaking diagnostic monitoring for problem resolution.

Implementation Comments

The current monalytics prototype is implemented in C/C++ using EVPath as its base communication layer. Our next implementation steps concern higher level facilities like those needed for leader election and like runtime structure reconfiguration (e.g., see our earlier work on iFlow [18] for ways to implement such functionality). We have not yet integrated monalytics with the vManage infrastructure described in [17], a first step in that work now underway being the placement of brokers into separate 'Management VMs' (virtual machines dedicated to carrying out monalytics functions) that can be deployed to machines at runtime when and if needed. This means that currently, the agents placed into Xen's dom0 (the 'driver domain') use statically created links to pre-created Management VMs containing brokers. Finally, the data capture methods used in monalytics exploit hypervisor level facilities like libvirt, system-level APIs like /proc, and select adhoc application-level monitoring. For generality, future work will deal with the use of monitoring standards like CIM, additional capture methods like those exploiting xentrace, and interfacing to higher level support for application-level monitoring provided by systems like Ganglia [19] or Tau [22].

3. THE NEED FOR MONALYTICS

The development and evaluation of monalytics principles, functionality, and infrastructure are driven by typical data center use cases. These cases and their motivation are described next. Additional detail appears in Section 4 of this paper.

Understanding Application and System Behaviors..

For large-scale data centers, it is critical to understand the dynamic behavior of infrastructure and systems, as well as of the applications running on data center hardware. An operator, for instance, will want to ascertain 'current health' by asking high level queries about certain system or machine states. Such queries go beyond simple questions about which machines currently appear to be up or down to questions that include the following: (1) are certain machine and subsystem (e.g., disk or network) utilizations within certain bounds, or (2) what is the current utilization of processors, memory, disk, network subsystems, or (3) what are the thermal and power draw values normalized by current system utilization, or (4) are certain application SLAs being met?

As evident from these questions, the data required for answering them must be captured at and traverse multiple levels of abstraction (e.g., platform sensors used for thermal or power draws vs. application-level measurements for ascertaining SLAs), and even conceptually simple questions like whether or not SLAs are being met will typically require captured data to be analyzed over certain time windows and for certain statistical likelihoods, perhaps even including economic or risk analyses [8]. Also evident is the fact that there is considerable parallelism in how such questions are answered, an example being the many entirely concurrent platform-local analysis actions that aggregate utilization values in conjunction with power draw and thermal data. At the same time, there is a necessity for global actions like aggregation in order to display results to operators and more importantly, to understand cross-cutting behaviors, perform historical analyses, and enter data in long term logs. Finally, the sources of questions like the above are not just human operators, but also automation engines like those performing datacenter provisioning, accounting, compliance checking, intrusion monitoring or spam detection, or similar continuous management tasks.

Questions like those above motivate several functionalities of monalytics. (1) Data-local actions are important to encourage parallelism in analysis. (2) Data reductions are enabled by combining data capture with analysis, as when reporting window averages rather than entire windows (e.g., to answer questions about current utilization) or when reporting bound violations for certain systems or applications. As stated earlier, both (1) and (2) are critical to attaining the scalability across time and length scales and levels of abstraction required for large-scale system management. (3) A final functionality is the need for dynamic control over what and how monitoring is performed, in conjunction with control over where decisions are made concerning the reactions to discovering certain phenomena: locally, globally, or both. An example is load balancing on a single multi-core platform to vacate as many cores as possible to reduce platform power draw, driven by temporary variations in application behavior and concurrency, coupled with additional, lower rate virtual machine movement across many platforms and racks for consolidation and power savings purposes, the latter perhaps driven by diurnal changes in load.

Experimental evaluations in Section 4 demonstrate the basic functionality of monalytics and establish the benefits of using source-near filtering or analysis. We also demonstrate our ability to dynamically control how and what monitoring is carried out.

Assessing and Managing Data Center Systems..

A typical issue in utility data center and cloud computing systems is misbehavior triggered by overloads, failures [4], or unusual application or system conditions. Such problems must typically be recognized and addressed quickly [9], in order to prevent congestion from worsening, failures from spreading to other subsystems, or the occurrence of catastrophic events like shutdowns. Continually assessing applications and systems, the paper's experimentation section evaluates the ability of monalytics to deal with a concrete scenario concerning failures. Three facts are of importance to monalytics in this context: (1) the ability to quickly recognize issues, by continuously and with low overhead checking relevant system or application behaviors on all nodes and for all application components involved; (2) the ability to link runtime detection to corrective actions, where 'local' linkages are key to reacting with low delay; and (3) the capability of dynamically moving from lightweight supervisory monitoring to detailed monitoring for problem diagnosis. Further, such actions must again be taken across multiple levels of abstractions, depending on the ability and methods used to detect overloads or failures.

Meeting Application Requirements..

Monalytics can be used to adjust and manage systems and applications in order to provide certain guarantees to end users, including response times, throughput levels, etc. It is typically not possible to implement such functionality with-

out operating across the multiple levels of abstraction and many subsystems existing in IT infrastructures. Further, since applications can be mapped onto different machines, including dynamically through runtime consolidation, monalytics must be capable of 'following' application components to wherever they run. Finally, recognizing the reasons why application requirements are not met can be quite complex. It may require detecting whether a system has failed or not, for example, and it typically requires global in addition to local analysis actions. Global analysis in turn requires monalytics to use aggregation trees and similar data dissemination structures to make appropriate data available to sophisticated methods for cause recognition and decision making.

The next section uses concrete instances of the general use cases described above to demonstrate and evaluate associated monalytics functionality.

4. EXPERIMENTAL EVALUATION

The experimental evaluation is divided into multiple subsections. The first subsection, describes the testbed, workload and target system used. The second subsection describes specific experimental scenarios and their use of monalytics features with results.

The experiments used in this section use a small testbed comprised of several multicore servers running the RUBiS [10] three-tier web services benchmark's servlet version on virtualized environment using the Xen hypervisor, with Apache, Tomcat, and MySQL each using different domUs. For some experiments we also use an open loop workload generator [24] and a simple load balancer [1]. Monitoring agents code run on each of the backend nodes. The agents, by default, monitor the cpu and memory utilization metrics of domUs and dom0.

Experimental Scenarios

Monalytics is evaluated in specific scenarios representing typical datacenter events or behaviors. The first two scenarios model misbehaviors due to software bugs and application misconfiguration, leading to corrective actions performed at different levels of abstraction. The third scenario demonstrates scalability achieved through data local analysis, using monalytics filtering functionality. Finally, we show how monalytics can be used in a large-scale decision analysis method [27]. In each of these scenarios, the cpu and memory utilization with monalytics infrastructure is less than 2% on brokers and broker leaders.

Runtime Component Misbehavior

This scenario demonstrates the usefulness of monalytics to 'assess and manage' data center systems. Here, we dynamically instantiate a monalytics control loop to deploy a corrective action, when monitoring and fault detection identify a faulty backend RUBiS instance.

We recreate an apache bug that causes segfaults and finally stops all interactions between certain RUBiS components. The effect of the bug is also observed at the loadbalancer's receive channel that interacts with the respective RUBiS instance. In this experiment, the behavior is detected by monitoring (1) the cumulative packets transmitted between apache, tomcat, and mysql domUs, and (2) the cpu utilization of RUBiS VMs along with the cumulative number of bytes received via the loadbalancer channels during the experiment run. Misbehavior is diagnosed when each of these metrics show no change in their values over some period of time.

Table 1: Impact Of Monalytics On End User Metrics

Case	Total Requests	Unsuccessful Requests
W/O Control Action	53535	13976
With Monalytics	52535	5763

For experimental runs, we inject one of the five RUBiS instances with the buggy Apache VM, generate workload requests at 1000requests/sec for 600sec, and start monitoring at 5 sec sampling rate.

With monalytics, fault detection is followed by a decision operation that sends an event to a trigger operator to trigger a corrective action, thus instantiating the aforementioned control loop. The action used simply restarts the Tomcat and Apache servers of the faulty RUBiS instance. Toward this end, we export libvirt's actuation APIs to start/stop a VM and then use the runtime code generation facilities of monalytics that calls these APIs.

Table 1 demonstrates the utility of associating simple analysis actions directly with data capture, as supported by monalytics. Utility is measured in terms of end user metrics – total unsuccessful requests over the period of the experiment run. During the run, once a fault is detected, analysis action recognize the application's need for corrective action, to prevent the Apache bug from unduly damaging application progress. This action is realized in the monalytics infrastructure, i.e., it is created via dynamic binary code generation and deployed on the faulty node. Similar in spirit to microreboots [9], it simply stops/starts the VM in question.

Table 1 reports the utility measured over the period of experiment run in two cases: (a) in worst case, if we do not take any corrective action and (b) using the monalytics-deployed corrective action. As known from prior work [9], such corrections help reduce the number of unsuccessful requests, in this case from 26% to 11%. More generally, the example demonstrates two important elements of the monalytics approach to system management: (1) runtime code generation and deployment of corrective actions, since one cannot assume prior knowledge of all actions that might be used in a large-scale system, and (2) the use of local vs. global control, that is, the placement of control actions 'near' where issues occur, to prevent undue monitoring traffic and encourage small delays between failure detection and reaction.

Performance Aware Load Balancing

This scenario demonstrates the use of continuous monalytics for to help applications meet their requirements (i.e., SLAs).

We emulate a degradation in the performance of critical bidding requests on one of the RUBiS instances servicing them. This is done by manipulating the maximum number of workers (MaxClients) and server limit (Server Limit) parameters on the Apache VM used by this instance. Manipulations cause longer processing time for bidding requests on this vs. other RUBiS instances, the effects of which are also observed at the loadbalancer's receive channel interacting with the instance. In the experiment, this behavior is detected by monitoring (1) the current number of busy workers and the request processing rate in each Apache VM and

Table 2: Impact Of Monalytics On End User Metrics

Case	Total Requests	Requests Meeting SLA
W/O Control Action	32213	21334
With Monalytics	32213	27763

(2) the cumulative number of bytes received via the loadbalancer channels during the experiment run along with the cpu utilizations of RUBiS VMs. Performance is considered degraded when these metrics consistently exhibit lower values for one RUBiS instance compared to others over some period of time.

There may be many causes for misbehavior, of course, but the objective of this experiment is not to identify failure causes but instead, to demonstrate the need for certain monalytics functionality. Specifically, monalytics must not only be able to to change how monitoring and analysis are done, but should also be able to dynamically inject simple management or corrective actions at different levels of abstraction and/or in different subsystems. In the previous experiment, corrective actions were taken by the hypervisor in monalytics leaf nodes, via VM stops/starts. In this experiment, actions are application-specific and are associated with the loadbalancer, i.e., in non-leaf nodes of monalytics topologies.

For these experiment runs, we inject misconfiguration on one of the three RUBiS instances serving bidding requests, generating workload at 1500requests/sec for 600sec. As the experiment proceeds, misbehavior detection causes the instantiation of a corrective action in the loadbalancer. The action simply flags the loadbalancer's degraded channel and then uses this information to change future load balancing decisions. This is implemented by modifying and exporting the loadbalancer API to mark up loadbalancing decisions and then again using runtime code generation to create codes on the loadbalancer node that call these APIs. We observe that such functionality can be implemented and changed separately from systems and application codes.

Table 2 reports the utility measured over the period of experiment run in two cases: (a) in the worst case, if we do not take any corrective action and (b) using monalytics to deploy a corrective action. Online monitoring and analysis via monalytics, coupled with the runtime installation of a simple control action, leads to an increase from 66% to 86% in the number of bidding requests that meet deadlines.

Scalability via Local Analysis

One way to attain scalability in monitoring is to immediately analyze data to the extent possible. A known useful purpose of local analysis is to use it to filter monitoring information, thereby reducing total data volumes passed across monalytics topologies. Runtime code generation and deployment can be used to attain this goal, as demonstrated next.

Data filtering is particularly important for high volume monitoring tasks like tracing. As an example, we trace the http requests processed by the Apache webserver. The trace record includes several fields, including: server_timestamp, request_url, request_params, request_time, client_ip, server_ip. In the experiment, we inject requests at 100 requests/sec for 600sec on a single webserver, and trace data is sent to the broker leader. The leader tracks the requests and their processing times and once a defined number of requests crosses some processing time threshold (200 ms), this triggers the decision to deploy a 'filter' operator at the apache webserver. This operator analyzes request trace data to send to the broker leader only those requests that have processing times above 200 ms and for such requests, it only forwards their url and running counts.

Without runtime filtering, tracing quickly leads to large monalytics volumes and overheads, resulting in 1.46 MB of request trace records for a 600 sec run. With filtering, the broker leader receives only 60.45KB of filtered data. This is because as with many such uses of tracing, in this experiment, only 2.3% of all requests exceed the stated processing time threshold, resulting in much data being transferred 'uselessly' when filtering is not used. More importantly, in actual systems, filtering criteria are dynamic, depending on current conditions and requirements.

Zoom-In Analysis

Building on the previous example demonstrating the importance of runtime data filtering, we next describe a more realistic set of decision methods used for this purpose. The goal is to demonstrate potential scalability via ***'failure-proportional'*** rather than *'system size-dependent'* monalytics actions.

The methods used here are based on our ongoing development of the EBaT lightweight, anomaly detection methods [27]. We again monitor the RUBiS application, where an agent in each machine's dom0 collects local virtual machine metrics (vcpu utilizations) and calculates a local entropy timeseries. These metrics, entropy timeseries, are then sent to a broker. Local entropy analysis results in low messaging overhead, because the metric transferred is a single entropy value of type float. The broker collects values provided by all agents and computes a global entropy timeseries. This ensures that any anomalies observed in the local entropy timeseries are reflected in the global timeseries, as well. A global entropy decomposition process is triggered after an anomaly in the global timeseries is detected. As the coordinator has the composite of each global entropy value, it can then 'zoom in' to the appropriate local entropy value/values that contribute to the abnormal global value change, thereby identifying the associated servers. Upon identification, additional monitoring and analysis are triggered in the appropriate local agents to further diagnose the anomaly, with sample values captured including application level data like number of busy threads, request rates, etc. This is done using a decision tree constructed using the machine learning method for system monitoring and failure diagnosis proposed in [12].

With zoom-in, rather than always monitoring all possible values of interest for detailed problem identification, monalytics' runtime methods for code deployment are used to dynamically install detailed monitoring only on where and when it is required. This means that monalytics overheads are proportional to the severity of failures – failure-proportio-nality – rather than being dependent on raw system or application size. To illustrate, consider the use of zoom-in analysis in large scale data centers, with a representative system considered in our work with EBaT [27] comprised of 81,920,000 virtual machines. Conservative estimates using say, one virtual machine to deploy one Apache server and observing 50 different monitoring metrics, results

in over 4G of monitoring data (assuming 10 bytes per metric per second) and in 40G bytes of data per second transferred to logging servers, generating undue data volumes and overheads. In contrast, when using the dynamic code generation and deployment capabilities of monalytics, runtime filtering operators use custom local decision trees for data-local analysis, moving select computations to data rather than moving excessive data amounts to analysis or logging nodes.

To demonstrate the feasibility of zoom-in analysis, a 10 hour experiment is measured for 3 hours, with 100 anomalies randomly injected into the RUBiS application. We compare a centralized method that (1) gathers all of the metrics from all agents, and (2) uses a typical threshold-based method for anomaly detection [27] with the proposed EBaT method. As shown in [27], EbAT outperforms the common threshold-based approach in terms of anomaly detection rate, false alarm rate, and accuracy. Similar results are obtained with the experiment run for this paper, but those are not shown for reasons of brevity. More important to this paper is the fact that with the zoom-in method, the monalytics overlay at runtime transfers a total of only 123.32 KB of local decision data, whereas almost three times as much data, 394.08KB, is transferred with the centralized solution. With offline analysis, for 10 hour runs, the centralized solution generates 1.15 MB of data against a 345.6KB data volume with data-local analysis. This again indicates that scalability can be attained by leveraging the dynamic and data-local processing capabilities of monalytics. Simple qualitative arguments for larger monalytics topologies can be used to further underline this argument.

5. RELATED RESEARCH

There are many partial or subsystem-level solutions to systems management. At one end of the spectrum, there are rich all-encompasing commercial monitoring and management solutions such as HP System's Insight Manager, IBM's Tivoli, and VMware's vCenter for data center environments. These systems perform centralized data collection and analysis, and provide some support for script-based triggering mechanisms. There is also hardware-level support focused on certain physical subsystems, such as HP's iLO or IBM's Director solutions for blade centers. None of the these solutions currently scale to the sizes needed in next generation data center systems.

There exist several open source tools for collecting monitoring data and for cluster-level monitoring [19]. [19] uses a hierarchical approach to monitoring where attributes are replicated within clusters using multicast methods and aggregated via a tree structure. Aggregation structures are evaluated in several related efforts, including [21,26,28]. [15] supports on-demand but not complex queries. [6] constructed on top of hadoop provides monitoring and analysis for large data-intensive codes and systems, focused on large volumes logs for failure diagnosis.

Note that most of the projects described above focus on scalability in data distribution and aggregation, supporting continuous or one-shot queries via certain hierarchical or peer-to-peer topologies, and they may use gossiping techniques or data structures like DHTs to access and distribute monitoring data. The monalytics approach to large-scale data center management can leverage the robustness and scale properties of such methods, but differs in also (1) combining data collection and aggregation with arbitrary analysis tasks, (2) permitting dynamic deployment and reconfiguration of monalytics graphs and operators, (3) providing scalability through data local analysis, and finally, (4) extending analysis with local management functions.

Monalytics leverages some of the concepts from earlier work on data streaming systems, including [5, 25], and our own research on the ECho publish/subscribe system [13] and the iFlow [18] event-based infrastructure, applied to high performance and to enterprise scale systems.

6. CONCLUSIONS AND ONGOING WORK

This paper presents the concept of monalytics along with the software architecture for realizing it in large scale data center environments. It provides concrete data center usage scenarios to establish the need for monalytics. Experimental evaluations demonstrate how simple actions integrated with monitoring can be helpful, particularly when it comes to issues of scale. We demonstrate the usefulness of monalytics through its features – deploying local control loops as corrective actions, installing them at different levels of abstraction or in different subsystems via monalytics leaf- or non-leaf nodes, and attaining scalability through data local analysis and failure proportionality for larger scale systems or tasks.

Ongoing work in this project includes (1) the integration of monalytics with the vManage [17] architecture, (2) its deployment in larger scale testbeds, including those provided by the OpenCirrus cloud computing infrastructure [2], and (3) additional experimental evaluations ranging from micro-benchmarks, to using parallel analysis operators, to demonstrating multiple distributed control loops in larger scale applications operating across different length and time scales.Also in progress are larger-scale experiments on the high performance machines accessible to our group [3].

Future work with monalytics concerns our ability to use and manage monalytics topologies, including dynamic topology configuration and the use of additional data aggregation and organizational methods, such as DHTs [26]. We are also exploring power-performance tradeoffs [17] in utility cloud computing systems, through detailed measurement and evaluation of commodity servers in an instrumented data center at Georgia Tech, including consideration of thermal and cooling issues.

7. REFERENCES

[1] Balance http://sourceforge.net/projects/balance/.

[2] Open Cirrus HP/Intel/Yahoo Open Cloud Computing Research Testbed https://opencirrus.org/.

[3] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng. DataStager: Scalable Data Staging Services for Petascale Applications. *HPDC*, 2009.

[4] S. Agarwala, F. Alegre, K. Schwan, and J. Mehalingham. E2EProf: Automated End-to-End Performance Management for Enterprise Systems. *DSN*, 2007.

[5] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive Control of Extreme-Scale Stream Processing Systems. *ICDCS*, 2006.

[6] J. Boulon, A. Konwinski, R. Qi, A. Rabkin, E. Yang, and M. Yang. Chukwa: A Large Scale Monitoring System. *Cloud Computing And Its Applications*, 2008.

[7] F. Bustamente, G. Eisenhauer, P. Widener, K. Schwan, and C. Pu. Active Streams: An Approach to Adaptive Distributed Systems. *HotOS-VIII*, May 2001.

[8] Z. Cai, Y. Chen, V. Kumar, D. S. Milojicic, and K. Schwan. Automated Availability Management Driven by Business Policies. *IM*, 2007.

[9] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - A Technique for Cheap Recovery. *OSDI*, 2004.

[10] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and Scalability of EJB Applications. *OOPSLA*, Nov 2002.

[11] H. Chen, G. Jiang, K. Yoshihira, and A. Saxena. Ranking the Importance of Alerts for Problem Determination in Large Computer Systems,. *ICAC*, 2009.

[12] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure Diagnosis Using Decision Trees. *ICAC*, 2004.

[13] G. Eisenhauer, F. Bustamente, and K. Schwan. Event Services for High Performance Computing. *HPDC*, 2000.

[14] D. Gupta, R. Gardner, and L. Cherkasova. XenMon: QoS Monitoring and Performance Profiling Tool http://www.hpl.hp.com/techreports/2005/HPL-2005-187.html.

[15] J.Liang, S.Y.Ko, I.Gupta, and K.Nahrstedt. MON: On-Demand Overlays For Distributed Systems Management. *Workshop on Real, Large Distributed Sytems*, 2005.

[16] E. Kiciman and B. Livshits. AjaxScope: A Platform for Remotely Monitoring the Client-side Behavior of Web 2.0 Applications. *SOSP*, 2007.

[17] S. Kumar, V. Talwar, V. Kumar, P. Ranganathan, and K. Schwan. vManage: Loosely Coupled Platform and Virtualization Management in Data Centers. *ICAC*, 2009.

[18] V. Kumar, Z. Cai, B. F. Cooper, G. Eisenhauer, K. Schwan, M. Mansour, B. Seshasayee, and P. Widener. Implementing Diverse Messaging Models with Self-Managing Properties using IFLOW. *ICAC*, 2006.

[19] M.L.Massie, B.N.Chun, and D.E.Culler. The Ganglia Distributed Monitoring System: Design, Implementation and Experience. *Parallel Computing*, 2004.

[20] J. H. Perkins, S. Kim, S. Larsen, et al. Automatically Patching Errors in Deployed Software. *SOSP*, 2009.

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

[22] T. Sheehan, A. Malony, and S. Shende. A Runtime Monitoring Framework for the TAU Profiling System. *International Symposium on Computing in Object-Oriented Parallel Environments*, December 1999.

[23] M. L. Simmons, A. H. Hayes, J. S. Brown, and D. A. Reed. *Debugging and Performance Tuning for Parallel Computing Systems*. IEEE Computer Society Press, 1992.

[24] C. Stewart, T. Kelly, and A. Zhang. Exploiting Nonstationarity For Performance Prediction. *Eurosys*, 2007.

[25] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An Information Flow Based Approach to Message Brokering. *International Symposium on Software Reliability Engineering*, 1998.

[26] S.Y.Ko, P. Yalagandula, I.Gupta, V.Talwar, D.Milojicic, and S.Iyer. Moara: Flexible and Scalable Group Based Querying Systems. *Middleware*, 2008.

[27] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan. Online Detection of Utility Cloud Anomalies Using Metric Distributions To appear. *NOMS*, 2010.

[28] P. Yalagandula and M. Dahlin. SDIMS: A Scalable Distributed Information Management System. *SIGCOMM*, 2004.