Non-intrusive, Out-of-band and Out-of-the-box Systems Monitoring in the Cloud

Sahil Suneja
University of Toronto
sahil@cs.toronto.edu

Canturk Isci
IBM T.J. Watson Research
canturk@us.ibm.com

Vasanth Bala
IBM T.J. Watson Research
vbala@us.ibm.com

Eyal de Lara
University of Toronto
delara@cs.toronto.edu

Todd Mummert
IBM T.J. Watson Research
mummert@us.ibm.com

ABSTRACT

The dramatic proliferation of virtual machines (VMs) in datacenters and the highly-dynamic and transient nature of VM provisioning has revolutionized datacenter operations. However, the management of these environments is still carried out using re-purposed versions of traditional agents, originally developed for managing physical systems, or most recently via newer virtualization-aware alternatives that require guest cooperation and accessibility. We show that these existing approaches are a poor match for monitoring and managing (virtual) systems in the cloud due to their dependence on guest cooperation and operational health, and their growing lifecycle management overheads in the cloud.

In this work, we first present *Near Field Monitoring* (NFM), our non-intrusive, out-of-band cloud monitoring and analytics approach that is designed based on cloud operation principles and to address the limitations of existing techniques. NFM decouples system execution from monitoring and analytics functions by pushing monitoring out of the targets systems' scope. By leveraging and extending VM introspection techniques, our framework provides simple, standard interfaces to monitor running systems in the cloud that require no guest cooperation or modification, and have minimal effect on guest execution. By decoupling monitoring and analytics from target system context, NFM provides "always-on" monitoring, even when the target system is unresponsive. NFM also works "out-of-the-box" for any cloud instance as it eliminates any need for installing and maintaining agents or hooks in the monitored systems. We describe the end-to-end implementation of our framework with two real-system prototypes based on two virtualization platforms. We discuss the new cloud analytics opportunities enabled by our decoupled execution, monitoring and analytics architecture. We present four applications that are built on top of our framework and show their use for across-time and across-system analytics.

SIGMETRICS'14, June 16–20, 2014, Austin, Texas, USA.

http://dx.doi.org/10.1145/2591971.2592009 .

Categories and Subject Descriptors

K.6.4 [**Management of Computing and Information Systems**]: System Management—*Centralization/ decentralization*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*; C.5.0 [**Computer System Implementation**]: General; C.4 [**Performance of Systems**]: Design studies

Keywords

Virtualization; Virtual Machine; Cloud; Data Center; Monitoring; Analytics; Agentless; VMI

1. INTRODUCTION

Cloud computing and virtualization technologies are dramatically changing how IT systems operate. What used to be a relatively static environment, with fixed physical nodes, has quickly transformed into a highly-dynamic environment, where (clusters of) virtual machines (VMs) are programmatically provisioned, started, replicated, stopped and deprovisioned with cloud APIs. VMs have become the processes of the cloud OS, with short lifetimes and a rapid proliferation trend [24].

While the nature of data center operations has changed, the management methodology of these (virtual) machines has not adapted appropriately. Tasks, such as performance monitoring, compliance and security scans, and product discovery amongst others are carried out using re-purposed versions of tools originally developed for managing physical systems or via newer virtualization-aware alternatives that require guest cooperation and accessibility. These approaches require a communication channel, i.e., a *hook*, into the running system, or the introduction of a software component, i.e., an *agent*, within the system runtime. There are two key problems with the existing approaches:

First, proliferation of VMs and their ephemeral, short-lived nature, makes the cost of provisioning and maintaining hooks and agents a major pain point. Moreover, the monitoring extensions "pollute" the end-user system, intervene with guest execution, and potentially open up new points of vulnerability. A recent observation from Amazon Web Services [3] highlights how agent operation and maintenance issues can impact managed systems. In this case, an incomplete maintenance update for DNS configuration in some of the agents, coupled with a memory leak issue led to a performance degradation for part of Amazon storage services.

This observation drives the first research question we address in our work: *How can we perform IT operations (monitoring, compliance, etc.) without relying on guest cooperation or in-VM hooks?*

Second, existing techniques work only so long as the monitored systems function properly, and they fail once a system becomes unresponsive—exactly when such monitoring information is the most valuable. A recent Google outage [12] presents a prime example to the effect of this limitation, where a significant portion of Google's production systems became unresponsive due to a dynamic loader misconfiguration. As a result of this, none of the in-system agents could publish data outside, neither was it possible to log in to the impacted systems for manual diagnosis. Thus,it was extremely difficult to get system information when it was most crucial. This observation drives our second research question: *How can we monitor and manage systems even when they become unresponsive or are compromised?*

To address these research challenges, this paper introduces *Near Field Monitoring* (NFM), a new approach for system monitoring that leverages virtualization technology to decouple system monitoring from system context. NFM extends VM introspection (VMI) techniques, and combines these with a backend cloud analytics platform to perform monitoring and management functions without requiring access into, or cooperation of the target systems. NFM *crawls* VM memory and disk state in an *out-of-band* manner from outside the guest's context, to collect system state which is then fed to the analytics backend. The monitoring functions simply query this systems data, instead of accessing and intruding each running system. In stark contrast with existing techniques, NFM seamlessly works even when a system becomes unresponsive (*always-on* monitoring), and does not require the installation and configuration of any hooks or agents in the target system (it works *out-of-the-box*). Unlike the in-VM solutions that run within the guest context and compete for resources allocated to the guest VMs, NFM is *non-intrusive* and does not steal guests' cycles or interfere with their actual operation. We believe our approach lays the foundation for the right way of systems monitoring in the cloud; very much like how we monitor processes in an OS today. NFM is better suited for responding to the ephemeral nature of VMs and further opens up new opportunities for cloud analytics by decoupling VM execution and monitoring.

There are three primary contributions of our work. First, we present our novel non-intrusive, out-of-band cloud monitoring and analytics framework, which we fully implement on real systems. It provides a service to query live as well as historical information about the cloud, with cloud monitoring and analytics applications acting as clients of this service. Our work presents how we can treat systems as documents and leverage familiar paradigms from the data analytics domain such as document differencing and semantic annotations to analyze systems. We also develop methods for low-latency, live access to VM's memory with optional consistency support, as well as optimizations that enable subsecond monitoring of systems. Second, we build several applications on top of our NFM framework based on actual enterprise use cases. These include (i) a cloud topology discovery and evolution tracker application, (ii) a cloud-wide realtime resource monitor providing a more accurate and holistic view of guests' resource utilization, (iii) an out-of-VM console-like interface enabling administrators to query system state without having to log into guest systems, as well as a handy "time travel" capability for forensic analysis of systems, and (iv) a hypervisor-paging aware out-VM virus scanner that demonstrates how across-stack knowledge of system state can dramatically improve the operational efficiency of common management applications like virus scan. Finally, we present a quantitative evaluation showcasing NFM's high accuracy, monitoring frequency, reliability and efficiency, as well as low impact on monitored systems.

The rest of the paper is organized as follows. Section 2 summarizes the techniques employed today for enterprise virtual systems monitoring. Section 3 gives a high level view of our solution architecture and discusses its benefits over existing alternatives. Section 4 gives the implementation details. Section 5 describes the applications we have built to demonstrate NFM's capability. Section 6 evaluates NFM's performance. Section 7 presents related work and Section 8 offers our conclusions.

2. EXISTING TECHNIQUES

System monitoring has been a major part of enterprise IT operations. We categorize the various techniques employed today as follows:

1. Application-specific in-VM agents for monitoring guest systems.
2. Application specific hooks for remotely accessing and monitoring systems.
3. Limited black-box metrics collected by the virtualization layer without guest cooperation.
4. General purpose agents or hooks that provide generic in-VM information through the virtualization layer.

Existing cloud monitoring and management solutions employ one or more of the above methods to deliver their services. For example, Amazon's CloudWatch [2] service falls under the third category in its base operation, while it can be extended by the end users with in-VM data providers (as in the first category) to provide deeper VM-level information. Other solutions such as Dell Quest/VKernel Foglight [14] is a combination of all three - guest remote access, in-VM agents, and hypervisor level metrics exported by VM management consoles like VMware vCenter and Red Hat Enterprise Management. To mitigate the limitations, and in particular, intrusiveness of custom in-VM techniques, an emerging approach has been the use of general purpose agents or *backdoors* inside the VMs that supply in-VM state information to the virtualization layer (fourth category). Various virtualization extensions such as VMware tools [48], VMSafe API [50], VMCI driver [47] and vShield endpoint [49] follow this approach. Other solutions also employ similar techniques and are summarized in Section 7.

There are important caveats with either of these options. First, in-VM solutions are only as good as the monitored system's operational health. This poses an interesting conundrum, where the system information becomes unavailable exactly when it is most critical—when the VM hangs, is unresponsive or subverted. Second, in-VM solutions face an uphill battle against the emerging cloud operation principles—ephemeral, short-lived VMs—and increasing VM proliferation. Their maintenance and lifecycle management has become a major pain point in enterprises. Furthermore,

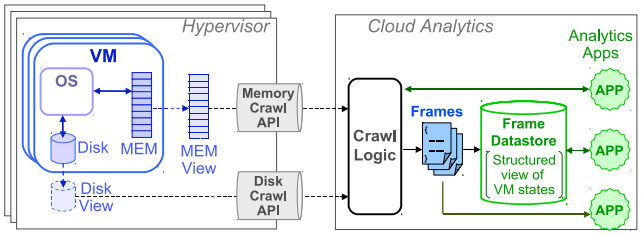

Figure 1: Introspection and analytics architecture

both agents and hooks modify the target systems, interfere with their operation, consume end-user cycles and are prime candidates for security vulnerabilities due to their external-facing nature. Third, even with generic-agent-based approaches, the problems of guest cooperation and intrusion do not completely disappear—although mitigated to some extent. However, these create a new challenge that goes against one of the key aspects of cloud computing: portability. Generic agents/hooks require specialization of VMs for the virtualization layer providing the monitoring APIs, which leads to vendor locking. Additionally, custom solutions need to be designed to work with each such agent provider.

3. NFM'S DESIGN

NFM's key differentiating value is its ability to provide monitoring and management capabilities without requiring guest system cooperation and without interfering with the guest's runtime environment. Architecturally, our overall framework, depicted in Figure 1, can be viewed as an introspection frontend and an analytics backend, which enables the VM execution - monitoring decoupling. The frontend provides an out-of-band view into the running systems, while the backend extracts and maintains their runtime state. The analogy is that of a Google like service running atop the cloud, enabling a query interface to seek live, as well as historical information about the cloud. Cloud monitoring and analytics applications then simply act as the clients of this service. We develop introspection techniques that build upon and extend traditional VMI approaches [20, 30], to gain an out-of-band view of VM runtime state. The analytics backend triggers VM introspection and accesses exposed VM state via *Crawl APIs*. Through these APIs, our *Crawl Logic* accesses raw VM disk and memory structures. VM disk state is accessed only once, before crawling a VM for the first time, to extract a small amount of information on the VM's OS configuration. *Crawl Logic* uses this information to access and parse raw VM memory to create a structured, logical view of the live VM state. We refer to this crawl output as a *frame*, representing a point-in-time view of the VM state. All frames of all VMs are stored in a *Frame Datastore* that the cloud monitoring and management applications run against, to perform their core functions, as well as to run more complex, cloud-level analytics across time or across VMs. By decoupling monitoring and management from the VM runtime, we enable these tasks to proceed without interfering with VM execution. The VMs being monitored are never modified in our approach, nor is the hypervisor or the VMM.

NFM is designed to alleviate most of the issues with existing solutions (Section 2). We follow three key principles to achieve this in a form that is tailored to the cloud operation principles:

1. **Decoupled Execution and Monitoring:** By decoupling target system execution from our monitoring and analytics tasks, we eliminate any implicit dependency between the two. NFM operates completely out of band, and can continue tracking system state even when a system is hung, unresponsive or compromised.
2. **No Guest Interference or Enforced Cooperation:** Guest context and cycles are precious and belong to the end user. Unlike most agent- or hook-based techniques that are explicitly disruptive to system operation, our design is completely non-intrusive. NFM does not interfere with guest system operation at all, nor does it require any guest cooperation or access to the monitored systems.
3. **Vendor-agnostic Design:** Our design is based on a generic introspection frontend, which provides standard crawl APIs. Our contract with the virtualization layer is only for base VMI functions—commonly available across hypervisors—exposing VM state in its rawest form (disk blocks and memory bytes). We do not require any custom APIs or backdoors between the VM and the virtualization layer in our design. All the data interpretation and custom intelligence are performed at the analytics backend, which also means simplified manageability as opposed to maintaining in-VM agents or hooks across several VMs.

In addition to alleviating some of the limitations of existing solutions, our design further opens up **new opportunities for cloud monitoring and analytics**. First, the decoupling of VM execution and monitoring inherently achieves *computation offloading*, where the monitoring / analysis computation is carried out outside the VM. This enables us to run some heavy-handed, *complex analytics*, such as full compliance scans with no impact on the actual systems. Second, many existing solutions actually track similar system features. By providing a singular data provider (backend datastore) for all analytics applications, we *eliminate redundancy* in information collection. Third, by shifting the scope of collected systems data from individual VMs to multiple cloud instances at the backend, we enable *across-VM analytics*, such as VM patterns, topology analysis, with simple analytics applications running against the datastore. Fourth, alongwith VM-level metrics, NFM is also exposed to host-level resource accounting measures, enabling it to derive a *holistic and true view* of VMs' resource utilization and demand characteristics.

NFM supports arbitrarily complex monitoring and analytics functions on cloud VMs with no VM interference and no setup requisites for the users. Our framework is designed to serve as the cornerstone of an *analytics as a service* (AaaS) cloud offering, where the users can seamlessly subscribe and unsubscribe to various out-of-the-box monitoring and analytics services, with no impact on their execution environments. These services span a wide range, from simple resource and security monitoring to across-the-cloud (anonymous) comparative systems analysis. Users would have the choice to opt into this service, paying for the cycles the hypervisor and the monitoring/analytics subsystem spends on behalf of them. One consideration for such a service is privacy for end users, however the guests already relinquish the same level of control to the existing agents running inside their systems, we only argue in favour of the same level of trust without the downside of having a potentially insecure foreign entity installed inside their runtime environment.

One limitation of NFM is a by-product of its reliance on VMI to crawl VM memory state, which involves interpreting kernel data structures in memory. These data structures may vary across OS versions and not be publicly documented for proprietary OSes. We discuss the tractability of this in our implementation discussion (Section 4.2). Also, as NFM focuses on the OS-level system view, application-level (e.g., a MapReduce worker's health) or architectural (e.g., hardware counters) state are not immediately observable via NFM unless exposed by the guest OS. Although, the hypervisor might allow measuring certain architectural states (e.g., perf counters) for a guest from outside.

4. IMPLEMENTATION

We implement NFM on two real-system prototypes based on two virtualization platforms, Xen and KVM. Currently we only support Linux guests, support for other OSes is discussed at the end of Section 4.2. Our overall implementation can be broken into four parts: (i) exposing VM runtime state, i.e., accessing VM memory and disk state and making them available over the crawl APIs; (ii) exploiting VM runtime state, i.e., interpreting the memory and disk images with the Crawl Logic to reconstruct the guest's runtime information; (iii) persisting this VM state information in the Frame Datastore for the subscribed applications; and (iv) developing high level applications to build on top of the extracted VM runtime state.

4.1 Exposing VM State

Most hypervisors provide different methods to obtain a view of guest VM memory. VMWare provides VMSafe APIs [50] to access guest memory. Xen [7] provides userspace routine (`xc_map_foreign_range`) via its Xen Control library (`libxc`) to re-map a guest VM's memory into a privileged VM. We use this technique to obtain a live read-only handle on the VM's memory. KVM [26] does not have default support for live memory handles. Although other techniques [8, 22] are able to provide guest memory access by modifying QEMU, our goal is to use stock hypervisors for generality. For KVM, options exist to dump VM memory to a file, via QEMU's `pmemsave` memory snapshotting, or `migrate`-to-file, or `libvirt`'s `dump`. While the VM's memory can be exposed in this manner, the overheads associated are non-trivial, as the guest gets paused during the dump duration. Therefore, for KVM, we develop an alternative solution to acquire a live handle on VM memory, while inducing negligible overhead on the running VM. As KVM is part of a standard Linux environment, we leverage Linux memory management primitives and access VM memory via QEMU process' `/proc/<pid>/mem` pseudo-file, indexed by the virtual address space backing the VM's memory from `/proc/<pid>/maps`.

We develop equivalent techniques for exposing VM disk state. We crawl VM disk state only once to collect required persistent system information that we use to interpret raw memory data structures (Section 4.2). We use standard filesystem methods to access and expose VM disks, which are represented as regular files on the host system.

In both prototypes, the obtained external view of raw VM memory and disk is wrapped and exposed as network-attached devices (FUSE over iSCSI) to the backend's *Crawl Logic* over common *Memory* and *Disk Crawl APIs*. Thus, the actual crawling and analytics components are completely decoupled from VM execution, and the upstream logic is transparent to the underlying virtualization technology.

Enhancements. (i) While our methods for exposing VM state generally have negligible latencies for most cloud monitoring and analytics applications, we build further optimizations to our techniques to curb down their latencies. Among these, one notable optimization is *selective memory extraction* for applications that rely on near-realtime information. The key insight behind this is that most applications actually require access to a tiny fraction of VM memory space, and therefore we can trade off completeness for speed. After an initial full crawl, we identify VM memory regions that hold the relevant information, and in subsequent iterations we opportunistically crawl only those sections. For dump-based handles, we batch relevant distributed memory regions together, to optimize the dump size and minimize dump cycles. Since some data structures like `task` lists, dynamically grow, we add additional buffering and resizing heuristics to accommodate these. Also, several distributed memory regions contained the relevant kernel data structure into small number of larger sized chunks. Overall, with such optimizations, we further reduce our overall latency impact by an order of magnitude. For example, for one of our realtime applications, *CTop*, this approach enables subsecond granularity realtime system monitoring even with heavyweight, dump-based methods, reducing overall latencies from few seconds to milliseconds.

(ii) We do not expect the guest OS to be in some steady state while extracting its live runtime state from outside the guest context. Since we avoid making any monitoring specific changes to the guest and not enforce any guest cooperation, lack of synchronization with the guest OS can potentially lead to inconsistency of views between the actual runtime state that exists inside the VM and what gets reconstructed from the view exposed outside, eg. a process terminating inside the guest while its memory mappings (`mm_struct`) are being traversed outside. To tackle this, we build additional routines for (optional) consistency support for KVM guest memory access via `ptrace()`- attach/detach on the QEMU container process. This trades off minor VM stuns for increased consistency. Our experiments show that even with a heavy (10 times/s) fork and kill workload consuming significant CPU, memory and network resources inside a VM, inconsistency of extracted state occurs not very often—in about 2% of crawler iterations, while extracting *full* system runtime state (Section 6.1).

4.2 Exploiting VM State

The backend attaches to the VM view exposed by the frontend, and implements the Crawl Logic that performs the logical interpretation of this raw state consisting of the disk block device and the memory byte array. Interpretation of the disk state is relatively well defined by leveraging standard filesystem drivers. The key challenge, however, is bridging the inherent semantic gap between the exposed raw VM memory state and the logical OS-level VM-internal view. The traditional in-VM approaches simply leverage guest OS context to extract system information, such as the `/proc` tree for process information. Our Crawl Logic achieves the same function by translating the byte-level memory view into structured runtime VM state. VM runtime information is distributed into several in-memory kernel data structures for processes (`task_struct`), mem-

ory mapping (`mm_ struct`), open files (`files_struct`), and network information (`net_devices`) among others. We overlay these `struct` templates over the exposed memory, and traverse them to read the various structure fields holding the relevant information [30]. To correctly map these data structures, we extract three important pieces of information from the VM disk image and/or a kernel repository:

1. **Version, Architecture and Map Parameters:** From the VM kernel log, we extract the running kernel version and the target VM architecture (32 vs. 64bit) for correctly sizing the data structures. Also read is the BIOS RAM map for determining the amount of VM memory to map and the size and layout of the VM memory regions.
2. **Starting Addresses for *structs*:** To identify the starting addresses for various kernel structures, such as initial task (`init_task`), module list (`modules`), and kernel page table (`init_level4_pgt`), we read the *System.map* file for the guest kernel, which includes the kernel exported addresses for these structures.
3. **Field Offsets:** After identifying entry points for kernel structures, we calculate the offsets to the relevant fields in these structures. We use the kernel source (or *vmlinux* build image) and the build configuration to determine offsets of desired fields.

Given a live VM memory handle and the above-mentioned pieces of information, the Crawl Logic's overall memory crawl process can be summarized as:

1. Reading a kernel exported address (X), such as the initial process' address (symbol `init_task`).
2. Mapping the associated data structure template (`struct task_struct`) to the memory region at address X.
3. Reading the relevant structure member fields after adding their relative offsets to the starting address X.
4. Continuing with the next object by reading its address via linked list traversal (`prev`, `next` member fields, together with offset arithmetic).

In addition to kernel structure traversal, we also traverse per-process page tables inside the extracted memory view, to translate a virtual addresses in a guest process' address space. We extract the process' page directory from its `mm_ struct`, and traverse its page tables. Our Crawl Logic can currently crawl all X86, X86_PAE, and x86_64 architectures, and also supports huge pages.

After introspecting all the relevant data structures, the Crawl Logic creates a single structured document (frame) for the crawled VM. This crawl document captures a rich set of information on VM state that the cloud monitoring and analytics applications can build upon. Our current frame format covers various VM features, including system, cpu, memory and process information, modules, address mappings, open files, network information and runtime resource use. This VM frame, with its corresponding timestamp and VM ID, is then loaded into the frame datastore described in the next section.

Discussion. (i) While we focus our backend implementation on Linux, NFM's applicability is not limited to a particular OS. Structure-offset extraction and VMI have been shown to work for Mac and Windows as well [5, 8, 34, 44]. Also, their versions are few as compared to Linux, and change slowly. (ii) The tractability of kernel data structure (DS) traversal based introspection solution is corroborated by prior studies reporting modest DS modifications across major Linux versions [37]. Indeed, our parsers for kernel versions 3.2 and 3.4.4 differ only in their network-device related DS among all of our tracked DS. Furthermore, the standardization trends in enterprise clouds also work in favor of this approach, limiting the OS-version variability. Even Amazon EC2 has a small number of base VM image types (5 different Linux OS versions). (iii) To portray NFM's generality, we have implemented our memory crawler to operate across a diverse configuration set- multi architecture (x86 / x86_64), multi core, variably (RAM) sized VMs running Linux kernels far apart in the version chain and from different vendors — RHEL6 / Linux2.6.32, Fedora14 / 2.6.35, Ubuntu12.04 / 3.2.0, Fedora16 / 3.4.4. (iv) In our experience, to generate structure-field information manually it takes a one-time effort of roughly an hour for a new OS version. This process can be automated by using crash [13] or gdb on debugging information enabled *vmlinux* kernel images. There also exist alternate OS-version independent memory introspection techniques (Section 7).

4.3 The Frame Datastore

The frame datastore is a repository of historical as well as live VM runtime states of all of the guest VMs, sent to it as structured frame documents by the "crawler" VM running the Crawl Logic. Although intended to be an actual database system for a full cloud deployment, in most cases we simply use the crawler VM file system as our current frame datastore. To ensure scalability with respect to the space overhead for maintaining VM state history (frames) for cloud-scale deployment, we control this by keeping incremental *delta frames* across successive crawls of a VM over time. Depending upon whether the delta is obtained over the base crawl or the most recent crawl, there are tradeoffs with regards to the ease of delta insertion/deletion vs. delta sizes, the former being trivial with base crawl deltas, while the latter being more manageable with latest crawl deltas. Section 6.7 evaluates this space overhead for maintaining system states across time.

4.4 Application Architecture

In our framework, the Cloud monitoring applications act as clients of the Frame Datastore, building on top of the rich system state it maintains throughout the VMs' lifetimes, to perform cloud analytics across time (past vs present system states) and space (across systems). Some of our target applications bypass the frame datastore for realtime operations (e.g. *CTop* resource monitoring), or for interfacing directly against the raw VM memory view (e.g. *PaVScan* virus scanning).

A benefit of visualizing system information as data documents (aka frames) is that it enables leveraging familiar paradigms from the data analytics domain such as diff-ing systems just like diff-ing documents and tagging systems with semantic annotations. This makes it easier to tackle modern day cloud management concerns arising from aggressive cloud expansion such as tackling system-drift— tracking how a deployed system with an initial desired state deviates over time— and performing problem diagnosis—drilling down into individual VM's to diagnose what caused the drift. Tagging point-in-time system states as being 'healthy' and

	TopoLog	CTop	RConsole	PavScan
Across-system analytics	●	●		
Across-time analytics	●		●	
Monitoring unresponsive or compromised systems		●	●	
Deep, across the stack system knowledge		●		●

Table 1: Key capabilities of our prototype applications

diff-ing them against faulty states, makes it easier and efficient to potentially troubleshoot system issues.

5. PROTOTYPE APPLICATIONS

Here we describe four concrete applications that we have built over our cloud analytics framework. These applications highlight NFM's capabilities and target some interesting use cases. Our fundamental principles for system analytics in the cloud are preserved amongst all applications: they all operate out-of-band with VM execution, are completely non-intrusive, and operate without requiring any guest cooperation.

Table 1 shows our four applications, *TopoLog*, *CTop*, *RConsole* and *PaVScan* and the key capabilities they highlight. *TopoLog* is a cloud topology discovery application that focuses on across-system analytics. It discovers VM and application connectivity by analyzing and correlating frames of cloud instances. It can also provide across-time analytics by tracing the evolution of cloud topology over frame history. *CTop* is a cloud-wide, realtime resource monitor that can monitor even unresponsive systems. CTop also showcases how deep, across-the-stack system information can provide more accurate and reliable system information. *RConsole* is an out-of-VM, console-like interface that is mainly designed with cloud operators in mind. Its pseudo-console interface enables administrators to query system state without having to log into guest systems, and even when the system is compromised. It also enables a handy "time travel" capability for forensic analysis of systems. *PaVScan* is a hypervisor-paging aware virus scanner, and is a prime example of how across-stack knowledge of system state, combining the in-VM view with the out-of-VM view, can dramatically improve the operational efficiency of common management applications like virus scan.

5.1 TopoLog

TopoLog is a network and application topology analyzer for cloud instances. It discovers (i) the interconnectivity across VMs, (ii) communicating processes inside each VM, (iii) connectivity patterns across high level applications, and (iv) (topology-permitting) per VM network flow statistics, without installing any hooks inside the VMs. TopoLog facilitates continuous validation in the cloud by ensuring that a desired topology for a distributed application (deployed as a pattern/set of VMs, e.g. via a Chef recipe [36]) is maintained. It detects unauthorized connections, bottleneck nodes and network resource use patterns. TopoLog offers other potential use cases, such as (i) optimizing inter- and intra-rack network use by identifying and bringing closer highly-communicating VMs, and (ii) simultaneous patching of interconnected VMs for minimal service downtime at the application level.

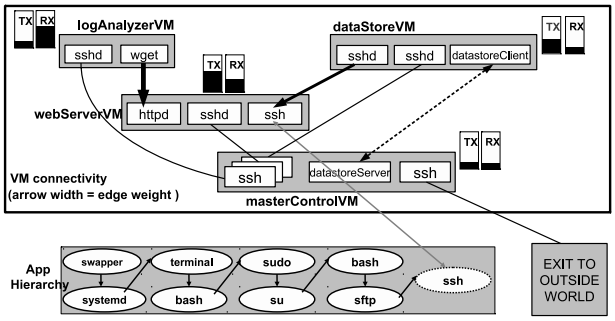

Figure 2: VM and app connectivity discovered by Topology Analyzer for 4 VMs

For each VM, topology analyzer extracts the per-process network information from the latest frame in the Frame Datastore. The extracted information (containing the socket type and state, associated source and destination IP addresses, and the process owning the socket) is correlated across all or a specified subset of cloud instances to generate a connectivity graph. We discover higher-level application information for the communicating processes by traversing the process tree within each VM frame. These steps are sufficient to generate the cloud network and application topology. In addition to these, TopoLog further discovers network traffic statistics for each VM by extracting and comparing counts for received, transmitted and dropped packets/bytes across two different timestamped frames.

Depending upon application knowledge and a particular topology, we can go one step further and estimate the weight of the connection edges in the topology graph. Since Linux does not maintain per process data transfer statistics, we estimate these by converting the connectivity graph to a linear system of equations. For a VM-to-VM connectivity graph of N VMs, we have at most $N^2 - N$ potential unknowns (graph edges) and $2 * N$ equations (per-VM system-wide received and transmitted bytes). If the number of equations are sufficient to solve for the actual number of edges, we can determine the weight of each connection. To bring down the number of unknowns, we also use domain knowledge such as information about connections with negligible or known constant weights.

Figure 2 shows one such connectivity graph generated automatically by the topology analyzer for an application pattern composed of 4 VMs. This application includes (i) a *MasterControl VM*, which monitors each application component and serves data to a DataStore VM; (ii) a *WebServer VM*, which serves client-side requests over http; (iii) a *DataStore VM*, which warehouses application data and receives updates from the MasterControl VM; and (iv) a *LogAnalyzer VM*, which downloads and analyzes the logs from the WebServer VM. As can be seen, the topology analyzer was able to discover all cluster connections such as the *masterControlVM* having long lived hooks into all other VMs over `ssh`, and feeding the *dataStoreVM* with data files. Also found was a connection that does not belong to the cluster, labeled as "exit to outside world". Also note that although the connection between *dataStoreVM* and *httpWebServerVM* is detected as an `ssh` connection, by traversing the process tree inside the latter's frame, we can get the higher level picture of this actually being a secure file transfer. A packet sniffing based topology discovery would not be able to detect intra-host connections for colocated VMs, or

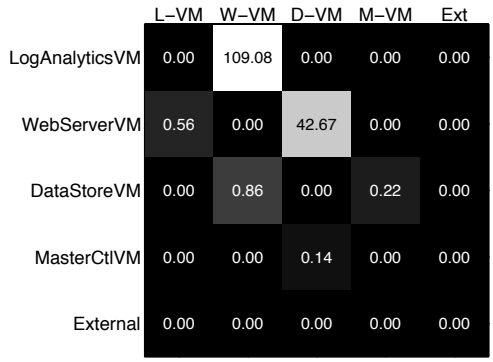

Figure 3: VM Connectivity Matrix [Mbps].

capture application-level information for the communicating systems.

Figure 3 further shows a snapshot of the derived network traffic statistics, as a *VM connectivity matrix* depicted as an intensity plot, for the same application pattern. An entry $M(r, c)$ in the matrix represents the rate of data transferred from VM_c to VM_r in Mbps. The row and column labels are identical (column labels are abbreviated in the plot). The connectivity matrix highlights the strongly-connected application components, which are the $\{WebServerVM \rightarrow LogAnalyzerVM\}$ and $\{DataStoreVM \rightarrow WebServerVM\}$ tuples in our case. This information is useful for both network-aware optimization of application resources and for continuous validation—by identifying unauthorized or abnormal application communication patterns.

5.2 CTop

CTop is a cloud-wide, realtime consolidated resource monitoring application. CTop is of equivalent fidelity and time granularity as the standard, in-band Linux `top` utility, with two enhancements for cloud-centric monitoring rather than traditional, system-centric techniques. First, modern cloud applications typically span across multiple VMs and hosts, requiring a distributed application-level view of resource use across system boundaries. CTop provides a single unified view of resource utilization across all applications and VMs distributed onto various physical machines within a cloud. Allowing for host and VM normalization (scaling), CTop dissolves VM and host boundaries to view individual processes as belonging to a single global cloud computer, with additional drill-down capabilities for restricting the view within hosts or VMs. Second, since CTop operates outside the VM scope, it is aware of both VM-level and hypervisor-level resource usage. Thus, it can provide a more accurate and holistic view of utilization than what the guest sees in its virtualized world. It appropriately normalizes a process' resource usage inside a VM to its corresponding usage on the host, or in terms of what the user paid for, for direct comparison of the overall application's processes across VMs. Equation 1 shows this normalization, where the actual CPU usage of a VM V's process P on host H is calculated in terms of the CPU usage of P inside the VM (CPU_V^P), overall CPU utilization of V (CPU_V^*), and the CPU usage of the VM on host H (CPU_H^V).

$$CPU_H^P = \frac{CPU_V^P}{CPU_V^*} \times CPU_H^V \quad (1)$$

To achieve realtime monitoring, CTop directly uses the crawler to emit a custom frame on demand at the desired monitoring frequency, bypassing the Frame Datastore. Fields of the frame include per-process PID, PPID, command, virtual and physical memory usage, scheduling statistics and CPU runtime. CTop analyzes frames from successive monitoring iterations to generate a *top*-like per-process resource usage monitor. Figure 4 compares the output of CTop with the standard in-VM *top* for a single VM. In this case, in VM measures and CTop measures match, as there is no host-level contention.

```
top - 11:58:42 up 1 day, 22:19,  1 user, load average: 0.90, 0.22, 0.11
Tasks: 57 total,  3 running,    54 sleeping,  0 stopped,    0 zombie
Cpu(s): 99.0%us, 0.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.7%hi, 0.0%si, 0.3%st
Mem:  2052104k total, 1976340k used,   75764k free,    3996k buffers
Swap: 6160380k total,  304068k used, 5856312k free,    1868k cached

  PID  USER   PR  NI  VIRT   RES  SHR  S  %CPU  %MEM   TIME+   COMMAND
 1942  root   20   0  1028m  1.0g  188  R  49.9  51.0  0:08.98  malloc
 1940  root   20   0  1028m  780m  136  R  49.5  38.9  0:11:91  malloc
    1  root   20   0  56220  1164  408  S   0.0   0.1  0:00.71  systemd
    2  root   20   0      0     0    0  S   0.0   0.0  0:00.00  kthreadd
```

```
Every 0.5s: ./topUpdate.sh

CPU up time: 4461430125 jiffies

  PID        VIRT          RES     %CPU   %MEM    TIME+     COMMAND
 1942   1052704KB    1047368KB    45.8   51.0   0:08:33   malloc
 1940   1052704KB     798816KB    45.8   38.9   0:11.92   malloc
    1     56220KB       1164KB     0.0    0.1   0:00.70   systemd
    2           0            0     0.0    0.0   0:00.00   kthreadd
    :
```

Figure 4: Above: in-VM *top*; Below: *CTop*

CTop's unified application-level resource utilization view allows for straightforward comparison of different instances of the same application across different VMs. This provides a simple form of problem diagnosis in the cloud, by tracking unexpected resource use variability among instances of an application. As we show in Section 6.3, CTop's holistic view of the operating environment helps explain performance and capacity conflicts that are hard to reason when monitoring is bound to the VM scope. We also evaluate CTop's latency and accuracy in Sections 6.1 and 6.2.

5.3 RConsole

RConsole is an out-of-band "console-like" interface for VMs being monitored by NFM. It is a read-only interface, with no side effects on the running instances. RConsole supports basic system functions such as `ls`, `lsmod`, `ps`, `netstat` and `ifconfig`. It is designed for cloud operators, to provide visibility into running instances without requiring access into the systems. RConsole runs against the system state captured in the frames indexed into the Frame Datastore. It implements a `sync` API call to crawl the current live state of a VM and retrieve its most up-to-date state, and a `seed` API to retrieve a prior stored state of a VM, which enables traveling back in time to observe past system state.

With RConsole, we perform simple security, compliance and configuration monitoring in an out-of-band fashion, without disrupting or accessing into the running systems. As RConsole operates by interpreting raw VM memory structures rather than relying on in-VM OS functions, it is also more robust against certain security attacks that may compromise a guest OS. We demonstrate this by infecting our VMs with the *AverageCoder* rootkit [18] that covertly starts and hides malicious processes, unauthorized users and open network connections from the guest OS. Figure 5 shows an example to this for network connections. Here the top box shows the (simplified) output of the standard in-VM `netstat` command, while the bottom box shows the output from

In-VM Console:

```
Active Internet connections (servers and established)
Proto Local Address        Foreign Address    State
tcp   127.0.0.1:25         0.0.0.0:*          LISTEN
tcp   9.XX.XXX.110:52019   9.XX.XXX.109:22    ESTABLISHED
:
tcp   9.XX.XXX.110:22      9.XX.XXX.15:49845  ESTABLISHED
```

RConsole:

```
Active Internet connections
Proto  Local Address       Foreign Address    State            PID Process
tcp    127.0.0.1:25        0.0.0.0:0          SS_UNCONNECTED   741 [sendmail]
tcp    9.XX.XXX.110:52019  9.XX.XXX.109:22    SS_CONNECTED    6177 [ssh]
:
tcp    9.XX.XXX.110:22     9.XX.XXX.15:49845  SS_CONNECTED   14894 [sshd]
tcp    0.0.0.0:2476        0.0.0.0:0          SS_UNCONNECTED 23304 [datacpy]
```

Figure 5: RConsole captures `datacpy`'s hidden listener connection

RConsole's `netstat`. Both outputs remain mostly similar, except for one additional entry in RConsole: a malicious `datacpy` process with a listening connection on `port 2476`. In-VM `netstat` fails to discover this as it relies on compromised guest OS exported functions, while RConsole can easily capture it from crawled VM state. More sophisticated attacks and VM introspection based counter-measures are well-established in prior studies [6, 20, 51].

RConsole is also greatly useful in troubleshooting system issues owing to its ability to travel back and forth in time. Once an anomaly is detected, it can trace back system state to detect the root cause, and compare across systems to identify similar good and bad system configurations. RConsole is even able to inspect a VM that we made completely dysfunctional by forcing a kernel panic; the entire runtime state still exists in the VM's memory, which RConsole is able to retrieve to pinpoint the culprit process and module. This ability to analyze unresponsive systems also plays a critical role in dramatically improving time to resolution in certain problems, as in the Google outage example of Section 1. A similar example to this has also been recently observed in one of our cloud deployments, where a network misconfiguration in one of the systems caused a serious routing problem in VM network traffic, rendering most of the VMs inaccessible. In both cases, RConsole's ability to perform simple troubleshooting operations (such as tracking network configurations via `ifconfig`) across time and across systems plays a critical role to pinpoint the offending systems and configurations in a simple and more efficient way.

5.4 PaVScan

PaVScan is a hypervisor paging [52] aware virus scanner that operates outside a VM's context, working directly on the raw VM memory state. We built PaVScan over the popular open source anti-virus project, ClamAV [11] and used its publicly available virus signature database. PaVScan searches for signatures of known viruses inside the VM's memory using the Aho-Corasick algorithm for pattern matching, and works by building and traversing a finite state machine from the signature database. Our use of the PaVScan application bypasses the Frame Datastore and interfaces with the memory crawler directly to get the live handle on the target VM's memory. Once VM memory is exposed, the virus signatures are directly scanned over the raw memory.

While one obvious main advantage of PaVScan is its ability to perform out-of-band virus scanning, this is not unique to this work [25]. The key differentiating aspect of our application is that it tracks hypervisor-paging—guest-agnostic reclaim of VM memory by the hypervisor. Using the crawler's interface to the hypervisor, PaVScan identifies which VM page frames are actually mapped on the physical RAM and which are paged out on disk. It then scans only the RAM-backed VM memory, while scanning the rest of the pages when they originally get paged out. This prevents unnecessary and costly page-ins from disk and ensures that the VM's working set does not get "thrashed" by the virus scan operation.

PaVScan presents a prime example of how deep, across-the-stack knowledge of system state can crucially impact across-system performance in the cloud. Traditional in-VM scanning techniques—for viruses or otherwise—are limited by their guest-only scope. Their actions, oblivious to the broader operating environment view, can be severely detrimental to both the system itself that they are monitoring and other instances sharing the cloud. In our case of virus scanning, a typical in-VM scanner(or even a paging-unaware out-of-VM solution) will indiscriminately scan all memory pages, as neither the guest nor the scanner are aware of any guest pages that have been paged out. Every such page access will cause a page-in from swap and potentially a page-out of a currently-loaded page (so long as the hypervisor doesn't give this VM more memory), severely impacting the performance of other applications on the VM. Depending on overall resource allocation and resource scheduler actions, this paging behavior can further impact other instances sharing the same memory and I/O resources in the cloud. In contrast, PaVScan's paging-aware, out-of-VM scanning approach operates with negligible impact on the monitored systems, while providing the same level of system integrity. We compare PaVScan with an in-VM scanner in Section 6.4.

6. EVALUATION

To evaluate NFM's performance, we use our cloud analytics applications to answer the following questions:

1. How frequently can we extract runtime VM information?
2. How accurate is our out-of-band VM monitoring?
3. Can we perform better than existing in-VM techniques with our holistic view of cloud operating environment?
4. How does out-of-VM monitoring improve operational efficiency in the cloud?
5. What is the overhead imposed on the VMs being monitored?
6. What is the impact of out-of-band monitoring on co-located VMs?
7. What is the space overhead of storing across-time (forensic) frame data in the datastore for each cloud instance?

Our experimental setup consists of physical systems with Intel Core-i5 processors, 4-64GB of memory and VT-x hardware virtualization support. These serve as the hypervisor hosts of our cloud environment, where our VM instances are deployed. The hosts run Linux 3.7.9, Xen 4.2.1 and QEMU-KVM 1.4. Our VMs run a variety of Fedora, RedHat and Ubuntu distributions (both 32 and 64 bit) in our experiments to ensure our crawlers work with a range of OS versions. Our analytics backend is run as a specialized VM with baked-in datastore and API components. The benchmarks used in our experiments are: `bonnie++` v1.96 [42], `x264` v1.7.0 [35], and `httperf` v0.9.0 [33]

6.1 Latency and Frequency of Monitoring

The amount of time it takes for our crawlers to extract runtime VM information varies with the richness of the de-

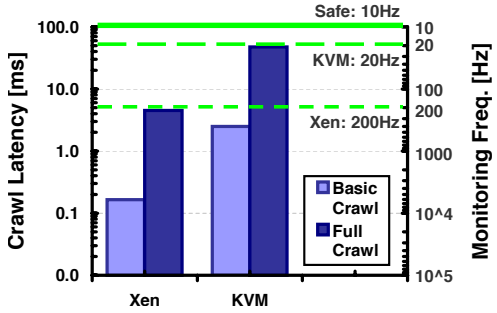

Figure 6: Measured crawling latencies and achievable monitoring frequencies (log scale).

sired information. Here we compare time required to extract *basic* process level resource use information for our `CTop` application, as well for deeper, *full* system information including all of system configuration, OS, module, process, file, CPU, memory, network connection data. All times are averaged over several runs while varying configured VM memory and CPUs, and the in-VM workloads emulating heavy process fork/kill behaviour and stressing the CPU, memory, disk and network (500 runs for each configuration).

For Xen, there is a one-time only operation for getting an initial handle on a target VM's memory (Section 4.1). This takes on average 0.37s per GB of VM memory. After this one-time operation, the crawler takes on an average 0.165 ms to extract the *basic* state and 4.5 ms for *full* state. For KVM, there is no separate memory handle acquisition step. The time taken to extract *basic* and *full* VM state information is 2.5 ms and 47.4 ms respectively. Figure 6 summarizes these results and highlights the corresponding achievable monitoring frequencies. As shown in the figure, full system crawls can be performed over 20 times/s (`Hz`) for KVM and 200 times/s for Xen (dashed horizontal lines). These times do not include the frame indexing time into the datastore, as this is off the critical path and is bypassed by the realtime monitoring applications, where latency matters. In either case, our results show that NFM can safely operate at a 10Hz monitoring frequency (solid horizontal line), which more than meets the practical requirements of most cloud applications.

The crawler has a negligible memory footprint and its CPU usage is proportional to the desired monitoring frequency and the number of VMs being monitored. In our KVM setup, for example, the crawler utilizing a full CPU core can monitor 1 VM at 20Hz or 20 VMs at 1Hz. Thus, there also exists a tradeoff between time granularity of monitoring and number of VMs that can be monitored in parallel.

Summary: Our crawlers extract full system state in less than 50ms, and our live out-of-VM monitoring can easily operate at a 10Hz frequency.

6.2 Monitoring Accuracy

We use our `CTop` application here to validate NFM's accuracy. We run a custom workload that dynamically varies its CPU and memory demand based on a configurable sinusoidal pattern. Figure 7 shows how well our remote monitor tracks the CPU resource use for a VM process with respect to *top*. The memory results are similar and omitted for brevity. The slight variation in measurements is due to the inevitable misalignment of the update cycles / sampling time points, as the two solutions operate asynchronously. Overall, our out-of-VM CTop monitoring is very accurate and reliable. The average sample variation between in-VM *top* and `CTop` metrics is very low, ranging between 4% and 1% at different time scales.

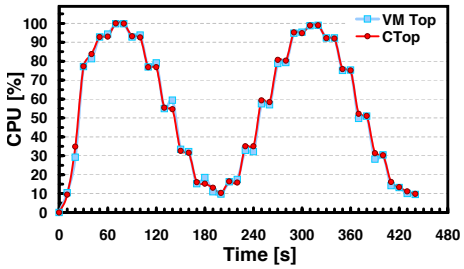

Figure 7: CPU utilization: in-VM `top` vs. `CTop`.

Summary: Our out-of-VM monitors accurately track VM process and resource usage information, providing the same level of fidelity and time granularity as in-VM techniques.

6.3 Benefits of Holistic Knowledge

As we previously discussed, NFM is privy to both in-VM and out-of-VM resource use measures. This unified, holistic view of systems enables significantly-better resource management in the cloud. We demonstrate the quantitative elements of this capability here with a webserver application, distributed across 3 identical instances, each running a LAMP stack. The VMs are facing high incoming HTTP load causing them to run at full utilization, the load originating from three separate `httperf` clients (one per server VM) generating identical request streams for 2MB files. While the VMs' configurations are identical, their current allocation of resources is not. Due to varying contention and priorities with respect to other colocated instances, the three VMs receive a 100%, 70% and 30% share of the CPU respectively. We demonstrate the holistic view of application performance characteristics with `CTop` and contrast this with the VM-only view of in-VM `top` in Figure 8.

Figure 8 shows the `httperf` statistics observed for each of the three VMs (top chart), the CPU utilization of the Apache, PHP and MySql processes inside each of these VMs as measured by *top* (middle chart) and as derived by `CTop` (bottom chart). As seen, *top*'s CPU utilization statistics look the same for all three VMs and thus cannot be used to reason for the different `httperf` sustained request rate, observed bandwidth and response times across the three VMs.

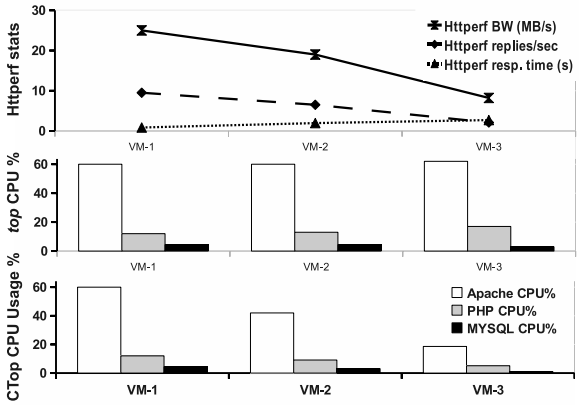

Figure 8: `top` vs. `CTop`: comparing LAMP processes across 3 VMs to explain `httperf` statistics.

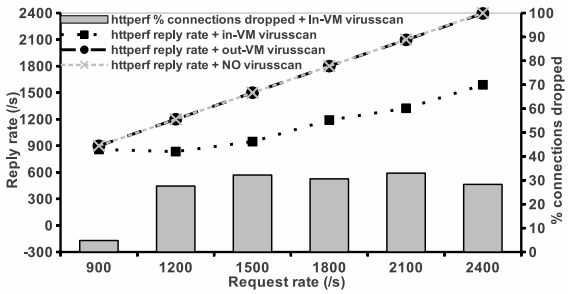

Figure 9: Httperf replyrate, connection drops vs Virusscan

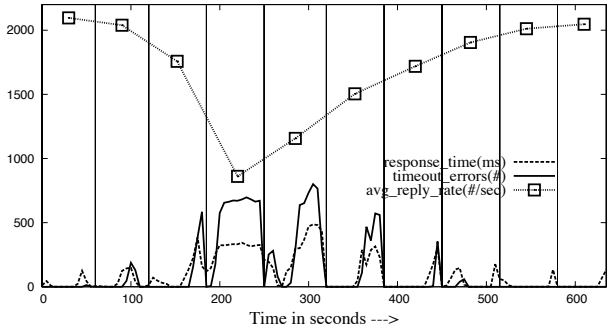

Figure 10: Httperf over 10 rounds (each bounded between successive vertical lines); Virusscan starts between round 2-3

However, by using `CTop`'s CPU utilization metrics (Section 5.2) that dissolve VM boundaries and normalize utilization of all of the LAMP processes across the application's instances at the host level, a clear CPU utilization difference can be spotted across the LAMP processes in different VMs. Thus, although the CPU utilization of the LAMP processes looks very similar when viewed inside the VMs in isolation, the true picture is in fact very different as captured by `CTop`. This clearly explains the application-level performance variability in `httperf` statistics comparing the true resource utilization across the application's distributed instances.

Summary: The unified, holistic view of the cloud environment enables accurate monitoring and interpretation of distributed application performance.

6.4 Operational Efficiency Improvements

We quantify the efficiency improvements achievable with NFM by evaluating virus scanning—representative of common scanning/inspection operations in the cloud—in our experimental framework. We use PaVScan as our out-of-VM scanner and install an identical in-VM scanner in the test VM, and compare their impact on the test VM's primary workload. We configure the VM with two VCPUs, where a web server is run as the primary application on one CPU and the in-VM virusscan application is run on a separate CPUs to avoid introducing any CPU contention. Hypervisor paging is enabled on the host via Xenpaging reclaiming 256MB from the VM's configured 1GB memory. A medium workload `httperf` server is setup on the VM with a working set of 256MB, from which it serves 2KB random content files to 3 different `httperf` clients running on 3 separate machines. The file size is chosen to be 2KB so that server is not network bound, and the working set size is chosen to be 256MB so that no guest level swapping is introduced. The average base-case runtime for the virus scanner was 17.5s to scan the entire VM memory. All `httperf` server statistics that follow are averaged over 5 runs.

Figure 9 shows the kind of reply rates that can be sustained by the webserver VM in this setup. Specifically, with the virusscanner turned off, the webserver VM is able to match an incoming request rate of upto 2400 requests/s without dropping any connections. PaVScan matches this reply rate very closely, reaching 2395 replies/sec with only 0.27% connection drops. On the other hand, `httperf` experiences a major performance hit with the in-VM scanner, where around 30% of connections are dropped with requests rates higher than 900 requests/s. Even with 900 requests/s, a drop rate of 5% is observed meaning that the actual sustainable rate is even below that. Effectively there is a decrease in performance by more than 63% (from 2400 to 900 serviced requests/s with in-VM scanning.). Additionally, response times degrade by an order of magnitude. The virusscanner's running time itself degrades to around 59s from 17.5s.

Furthermore, the performance impact of the in-VM virus scanner lasts much longer than just the scan duration. Figure 10 shows that it takes a much longer time for `httperf` to recover after the in-VM virusscanner destroys `httperf`'s file cache due to swapped page-ins. Shown are 10 `httperf` rounds (bounded by vertical lines in the figure) of servicing 256MB worth file requests each, fired at a rate of 2100 requests/s. The in-VM virusscanner starts around between rounds 2-3 and completes by round 5. As can be seen it takes about 7 rounds (460 seconds) for `httperf` to re-establish its file cache for sustaining the incoming request rate. For the entire 10 rounds, there is an 18.6% performance degradation in terms of serviced requests per second and 12.5% connection drops. Thus, in this hypervisor paging scenario, the in-VM scanner's impact is felt even long after it exits. In contrast the out-of-VM scanner shows only negligible impact on the guest during and after its execution.

Summary: The ability to push common cloud operations out of the VMs' scope can lead to dramatic improvements in system performance compared to their in-VM counterparts.

6.5 Impact on VM's Performance

We measure NFM's overhead on a target system's workload with three experiments. We expose the live VM state to (i) monitor the VM with `CTop`, (ii) hash the VM's memory, and (iii) scan the VM's memory with `PaVScan`. All experiments are repeated 5 times. The target VM and workload setup are similar to those in Section 6.4 (except hypervisor paging turned off). We use two workload configurations for the webserver VM: (i) A 256 MB working set to avoid guest swapping, serving all requests from memory without accessing the disk, and (ii) a 512 MB working set that involves the standard *kswapd* swap daemon. The `httperf` server is pushed to CPU saturation in both cases. The host system is never overloaded in the experiments. Figure 11 shows the performance impact on the VM for both workload configurations, when the three described out-of-VM monitoring applications are run for the same VM.

Realtime Monitoring: We measure the impact of monitoring the webserver VM at a 10Hz frequency, while extracting the *full* system state (Section 6.1) in addition to `CTop`'s per process state, at each monitoring iteration. No drop is recorded in the VM's serviced request rate, but the response time degrades by about 2%, only for 256 MB working set.

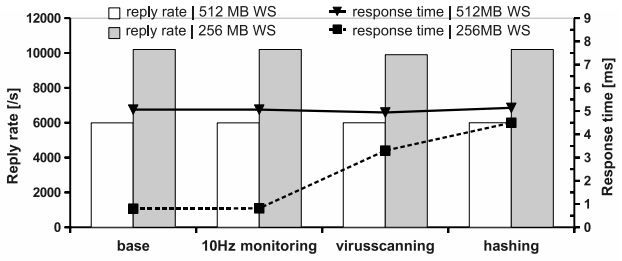

Figure 11: Impact on webserver VM with parallel out-of-band monitoring and management

Hashing VM's Memory: As a stress-test benchmark for memory crawling, we hash all of the VM's memory pages with *Mhash* library's MD5 hashing [31]. Even this benchmark has no visible impact on the VM's `httperf` server capacity as it continues to sustain the original request rate. However, the average response time degrades from 0.8 ms/request to 4.5 ms/request for the 256 MB working set scenario, while remaining within the timeout threshold. For the 512 MB working set, the overhead is merely 1.5%.

VM Memory Scanning: Our virus scanner prototype, which is close to a worst-case application, introduces a 2.9% degradation on the `httperf` sustainable request rate with an average response time of 3.3ms for the 256 MB working set. In this case, the VM runs at its absolute limit, continuously serving requests directly from memory. Interestingly, the higher `httperf` working set size of 512MB (involving *kswapd*) records no impact on the `httperf` server capacity with PaVScan running simultaneously, as the application occasionally requires new page-ins, which is the common case for most practical applications.

Summary: The target VMs are not heavily impacted in the common operating case of realtime monitoring. The negative impact observed with heavy handed operations needs further analysis.

6.6 Impact on Co-located VMs

Here, we measure the performance overhead introduced on a VM (X) while other co-located VM's on the same host are being monitored. Since the VM X itself is not being monitored, any impact on its performance can be attributed as a side effect of monitoring the other colocated VMs. We use Xen for a clean administrative domain-guest domain separation for this experiment. We run the memory crawler inside `Dom0` itself, and monitor VMs at 10Hz, extracting the *full* system state from the VM at each iteration. Alongside `Dom0`, three other VMs run on their separate cores on a quad-core host. We measure the impact on each VM separately, while monitoring the other two VMs. The 3 VMs are put under stress, each running a different workload: (i) CPU-bound `x264` video encoding benchmark, (ii) disk-bound `bonnie++` disk benchmark, and (ii) a full-system stress workload simulated by an `httperf` webserver configured exactly as described in Section 6.5. By virtue of Xen's frontend/backend driver model, `Dom0` becomes responsible for arbitrating VMs' access to network and disk. Disk caching at hypervisor is disabled so that true disk throughputs can be measured. The host system (including `Dom0`) itself is never overloaded for repeatable and consistent measurements.

While monitoring the remaining 2 VMs at a 10Hz frequency, and repeating each experiment 5 times, neither the `bonnie++` VM nor the `x264` VM see any impact on their read/write throughputs and framerate respectively, but the `httperf` server VM's maximum sustainable request rate drops by 2.2%. For the latter, as in Section 6.5, relaxing the artificial "no guest swapping" constraint and thus increasing the working set size from 256MB to 512MB, also results in no impact on VM performance. An important point to further note is that a 10Hz frequency is actually high for most cloud monitoring and analytics operations, and the crawler's impact can be further minimized by operating at lower but still largely practical frequencies at 1Hz or less. Moreover, our execution - monitoring decoupled design favours these operations to be handed off to other lightly loaded or dedicated hosts, further minimizing crawler impact.

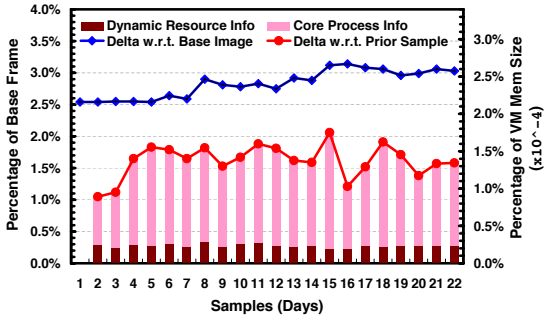

Figure 12: State management overhead with delta frames.

Summary: NFM is lightweight and does not have a heavy monitoring side-effect on the host's colocated VMs.

6.7 Space Overhead

Here we evaluate the storage requirements for maintaining VM state history in our frame datastore to enable across-time system analytics. We use the two delta frame extraction approaches described in Section 4.3. Shown in Figure 12 are the delta frame sizes relative to the base frame and relative to the previous frame, for a daily crawl of a VM over a 3-week period, while running various routine tasks and additional end-user applications. When computing deltas over the base fame (top curve) the frame sizes grow from 2.5% to 3% of the base frame size. However, when deltas are computed over the most-recent crawled state, the frame sizes do not grow over time (lower curve), averaging around 1.5% of the full frame size. Also, as can be seen, the amount of information that we need to keep for each VM for across-time analysis is minuscule compared to the actual VM sizes (delta frame sizes are only 0.00015% of the VM's 4GB memory size). In our experiments, the full frame sizes vary around 300KB-400KB and the delta frames are about 3KB-8KB.

If we scale these numbers out for a hypothetical cloud deployment with 10,000 VMs, the overall space overhead for our datastore with daily system snapshots and a year-long time horizon amounts to 14GB to 33GB, which is quite manageable even without any potential across-VM optimizations.

Summary: The overall space overhead of maintaining VM state across all instances and across time is manageable and is potentially scalable to large-scale cloud deployments.

7. RELATED WORK

Memory Introspection Techniques. Previous studies explored various methods for exposing VM memory state for introspection. Some of these employ hardware methods, such as direct memory access [1, 32] or PCI hooks [9]. While

others rely on software techniques, including in-VM kernel modules (Volatilux [17]), in-VM memory mapping (i.e., via `/dev/mem`), VMM-level memory mapping / snapshotting, and remapping of guest address space inside a trusted VM (XenAccess [38], and LibVMI [8]). Prior work on interpreting the memory data structures describe different approaches with different requisites and degrees of inference. IDetect [30] interprets memory by accessing OS-specific, predetermined offsets to extract specified system information. Other techniques for automatically detecting the offsets also exist, which rely on pattern matching, function disassembly heuristics, guest support or additional kernel debug information [17, 27, 4, 10, 28, 13]. Recent work has also proposed automated introspection solutions that do not require detailed knowledge of OS kernel internals. Virtuoso [15] collects in-VM training code traces to create out-of-VM introspection code. VM Space Traveler [19] uses instruction monitoring at VMM layer to identify and redirect introspection related data to guest OS memory. Our framework leverages VMM memory mapping or snapshotting capabilities for our two prototypes, and our crawling approach shares some aspects of IDetect. We use this introspection framework, however, for fundamentally different applications- non-intrusive, out-of-band cloud monitoring.

Memory Introspection Applications. Most previous VM introspection work focuses on security, digital forensics, malware and intrusion detection applications [21, 43, 50]. Garfinkel and Rosenblum [20] use memory introspection for intrusion detection, with detectors for tracking program integrity, virus signatures and network sockets. Other work employs introspection techniques for detecting rootkits, identifying hidden drivers and anti-malware solutions [16, 19, 25]. Outside the security domain, IBMon [40] comes closest to our approach, using memory introspection to estimate bandwidth resource use for VMM-bypass network devices.

Other Cloud Monitoring Techniques. In addition to the ones already discussed, other existing VM monitoring techniques can also be viewed as falling into one or more categories described in section 2. PHD Virtual's [39] basic VM-as-blackbox metrics use only hypervisor level information, while in depth VM level metrics requires running scripts / installing 'intelligent agents' inside the VMs. Reflex vWatch monitoring [41] uses information from VMware vCenter. VMware vCenter Operations Management Suite [45] is also a combination of hypervisor level metrics, together with in-guest agent (vFabric Hyperic). VMware VIX API [46] uses VMware Tools to track VM information and actuate actions through this interface. Several security solutions such as McAfee MOVE, TrendMicro DeepSecurity, Reflex vTrust, SADE, CloudSec use VMSafe / vShield single agent approach.

Some exceptions include: (i) Hypertection [23], that also calls its security solution agentless, it can access memory of Hyper-V VMs only but their approach is unknown, and (ii) Litty and Lie's out-of-VM patch auditing [29] that uses architectural/ hardware introspection to be guest OS-agnostic. They monitor virtual hardware information (page bits, cpu registers) that exists at the hypervisor level to detect execution of unpatched binaries and non-binary files, given a database of known binaries and patches. A departure from the usual memory introspection way of inferring VM's runtime state, it is limited in the kind of information that can be recovered while operating at the virtual hardware level and relies on a functional VM environment.

8. CONCLUSION

In this paper we propose *Near Field Monitoring*, a fundamentally different approach to systems monitoring and analytics in the cloud. We show that traditional in-VM techniques or newer virtualization-aware alternatives are not a good fit for modern data centers, and address their limitations with our non-intrusive, out-of-band cloud monitoring and analytics framework. NFM decouples system execution from monitoring and analytics functions, and pushes these functions out of the systems' scope. The NFM framework provides "always-on" monitoring, even when the monitored systems are unresponsive or compromised, and works "out-of-the-box" by eliminating any need for guest cooperation or modification. We describe the implementation of NFM across two virtualization platforms, and present four prototype applications that we built on top of our framework to highlight its capabilities for across-systems and across-time analytics. Our evaluations show that we can accurately and reliably monitor cloud instances, in realtime, with minimal impact on both the monitored systems and the cloud infrastructure. Our ongoing work with NFM involves deploying our framework in a real production data center to evaluate its scalability characteristics, to explore new potential applications, and to discover emerging patterns across large-scale system data.

We believe our work lays the foundation for the right way of systems monitoring and analytics in the cloud. With its ability to support non-intrusive, fine-grain monitoring and complex analytics, NFM serves as the the cornerstone of an "analytics as a service (AaaS)" cloud offering, where the end users of the cloud can seamlessly subscribe to various out-of-the-box monitoring and analytics services, with no impact or setup requisites on their execution environments.

9. ACKNOWLEDGEMENTS

We would like to thank our anonymous reviewers and our shepherd Thu Nguyen for their helpful suggestions on improving this paper. This work is supported by an IBM Open Collaboration Research award.

10. REFERENCES

[1] Adam Boileau. Hit by a Bus: Physical Access Attacks with Firewire. *RuxCon* 2006. `http://www.security-assessment.com/files/presentations/ab_firewire_rux2k6-final.pdf`.

[2] Amazon. CloudWatch. `http://aws.amazon.com/cloudwatch/`.

[3] Amazon. Summary of the October 22,2012 AWS Service Event in the US-East Region. `https://aws.amazon.com/message/680342/`.

[4] Anthony Desnos. Draugr - Live memory forensics on Linux. `http://code.google.com/p/draugr/`.

[5] M. Auty, A. Case, M. Cohen, B. Dolan-Gavitt, M. H. Ligh, J. Levy, and A. Walters. Volatility - An advanced memory forensics framework. `http://code.google.com/p/volatility`.

[6] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu. DKSM: Subverting Virtual Machine Introspection for Fun and Profit. In *SRDS*, pages 82 –91, 2010.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, 2003.

[8] Bryan Payne. Vmitools - An introduction to LibVMI. `http://code.google.com/p/vmitools/wiki/LibVMIIntroduction`.

[9] B. D. Carrier and J. Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, 2004.

[10] A. Case, L. Marziale, and G. G. RichardIII. Dynamic recreation of kernel data structures for live forensics. *Digital Investigation*, 7, Supplement(0):S32 – S40, 2010.

[11] ClamAV. Clam AntiVirus. `http://www.clamav.net`.

[12] C. Colohan. The Scariest Outage Ever. CMU SDI/ISTC Seminar Series, 2012.

[13] David Anderson. White Paper: Red Hat Crash Utility. `http://people.redhat.com/anderson/crash_whitepaper/`.

[14] Dell Quest/VKernel. Foglight for Virtualization. `http://www.quest.com/foglight-for-virtualization-enterprise-edition/`.

[15] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *IEEE Security and Privacy '11*, pages 297–312.

[16] B. Dolan-Gavitt, B. Payne, and W. Lee. Leveraging forensic tools for virtual machine introspection. Technical Report GT-CS-11-05, Georgia Institute of Technology, 2011.

[17] Emilien Girault. Volatilitux- Memory forensics framework to help analyzing Linux physical memory dumps. `http://code.google.com/p/volatilitux/`.

[18] M. F. Linux Rootkit Implementation. `http://average-coder.blogspot.com/2011/12/linux-rootkit.html`, 2011.

[19] Y. Fu and Z. Lin. Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *IEEE Security&Privacy'12*.

[20] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *NDSS*, pages 191–206, 2003.

[21] B. Hay and K. Nance. Forensics examination of volatile system data using virtual introspection. *SIGOPS Oper. Syst. Rev.*, 42(3):74–82, 2008.

[22] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with OSck. In *ASPLOS*, pages 279–290, 2011.

[23] Hypertection. Hypervisor-Based Antivirus. `hypertection.com`.

[24] Jack of all Clouds. Recounting EC2 One Year Later. `www.jackofallclouds.com/2010/12/recounting-ec2/`.

[25] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based out-of-the-box semantic view reconstruction. In *CCS '07*, pages 128–138.

[26] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux Virtual Machine Monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, 2007.

[27] I. Kollar. Forensic RAM dump image analyser. Master's Thesis, Charles University in Prague, 2010. `hysteria.sk/~niekt0/fmem/doc/foriana.pdf`.

[28] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures. In *NDSS*, 2011.

[29] L. Litty and D. Lie. Patch auditing in infrastructure as a service clouds. In *VEE '11*.

[30] Mariusz Burdach. Digital forensics of the physical memory. 2005. `http://forensic.seccure.net/pdf/mburdach_digital_forensics_of_physical_memory.pdf`.

[31] N. Mavroyanopoulos and S. Schumann. Mhash. `http://mhash.sourceforge.net`.

[32] Maximillian Dornseif. 0wned by an iPod. *PacSec Applied Security Conference* 2004. `http://md.hudora.de/presentations/firewire/PacSec2004.pdf`.

[33] D. Mosberger and T. Jin. httperf - a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, 1998.

[34] Nirsoft. Windows Vista Kernel Structures. `http://www.nirsoft.net/kernel_struct/vista/`.

[35] OpenBenchmarking/Phoronix. x264 Test Profile. `http://openbenchmarking.org/test/pts/x264-1.7.0`.

[36] Opscode. Chef. `http://www.opscode.com/chef/`.

[37] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in linux device drivers. In *EuroSys'06*.

[38] B. Payne, M. de Carbone, and W. Lee. Secure and Flexible Monitoring of Virtual Machines. In *Twenty-Third Annual Computer Security Applications Conference*, pages 385 –397, 2007.

[39] PHD Virtual. Virtual Monitoring. `http://www.phdvirtual.com/`.

[40] A. Ranadive, A. Gavrilovska, and K. Schwan. Ibmon: monitoring vmm-bypass capable infiniband devices using memory introspection. In *HPCVirt*, pages 25–32, 2009.

[41] Reflex. vWatch Monitoring. `http://www.reflexsystems.com/Products/vWatch`.

[42] Russell Coker. Bonnie++. `http://www.coker.com.au/bonnie++/`.

[43] A. Srivastava and J. Giffin. Tamper-Resistant, Application-Aware Blocking of Malicious Network Connections. In *RAID*, pages 39–58, 2008.

[44] S. Thomas, K. Sherly, and S. Dija. Extraction of memory forensic artifacts from windows 7 ram image. In *IEEE ICT '13*, pages 937–942, April 2013.

[45] VMware. vCenter Operations Management Suite. `http://www.vmware.com/products/vcenter-operations-management/`.

[46] VMware. VIX API Documentation. `http://www.vmware.com/support/developer/vix-api/`.

[47] VMware. VMCI Overview. `http://pubs.vmware.com/vmci-sdk/`.

[48] VMware. VMWare Tools. `http://kb.vmware.com/kb/340`.

[49] VMware. vShield Endpoint. `http://www.vmware.com/products/vsphere/features-endpoint`.

[50] VMWare Inc. VMWare VMSafe security technology. `http://www.vmware.com/company/news/releases/vmsafe_vmworld.html`.

[51] S. Vogl. A bottom-up Approach to VMI-based Kernel-level Rootkit Detection. PhD Thesis, Technische Unversitat Munchen., 2010.

[52] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.