# Time Capsule: Tracing Packet Latency across Different Layers in Virtualized Systems

Kun Suo Jia Rao
The University of Texas at Arlington
kun.suo@mavs.uta.edu
jia.rao@uta.edu

Luwei Cheng
Facebook
chengluwei@fb.com

Francis C. M. Lau
The University of Hong Kong
fcmlau@cs.hku.hk

## Abstract

Latency monitoring is important for improving user experience and guaranteeing quality-of-service (QoS). Virtualized systems, which have complex I/O stacks spanning multiple layers and often with unpredictable performance, present more challenges in monitoring packet latency and diagnosing performance abnormalities compared to traditional systems. Existing tools either trace network latency at a coarse granularity, or incur considerable overhead, or lack the ability to trace across different boundaries in virtualized environments. To address this issue, we propose *Time Capsule* (TC), an in-band profiler to trace packet level latency in virtualized systems with acceptable overhead. TC timestamps packets at predefined tracepoints and embeds the timing information into packet payloads. TC decomposes and attributes network latency to various layers in the virtualized network stack, which can help monitor network latency, identify bottlenecks, and locate performance problems.

## 1. Introduction

As virtualization has become mainstream in data centers, a growing number of enterprises and organizations are moving applications into the cloud, such as Amazon EC2 [1]. It is well believed that virtualization introduces significant and often unpredictable overhead to I/O-intensive workloads, especially latency-sensitive network applications. To improve user experience and guarantee QoS in the cloud, it is necessary to efficiently monitor and diagnose the latency of these applications in virtualized environments.

However, compared to traditional systems, it is more challenging to trace I/O workloads and troubleshoot latency problems in virtualized systems. First, virtualization introduces additional software stacks and protection domains while traditional tracing tools designed for physical machines are unable to trace across the boundaries (e.g., between the hypervisor and the guest OS). Second, workload consolidation inevitably incurs interference between applications, which often leads to unpredictable latencies. Thus, fine-grained tracing of network latency, instead of coarse-grained monitoring, is particularly important to guaranteeing QoS. Third, many applications in the cloud are highly optimized and the tracing tool should incur negligible performance impact [31]. Otherwise, tracing overhead may hide seemingly slight but relatively significant latency changes of the cloud services. Last, as clouds host diverse workloads, it is prohibitively expensive to devise application specific tracing mechanisms and this calls for a system level and application transparent tracing tool.

There exist many studies focusing on system monitoring and diagnosing. Traditional tools, such as SystemTap [12], DTrace [17], Xentrace [13], or existing instrumentation systems, such as DARC [34] and Fay [21], are limited to use within a certain boundary, e.g., only in the hypervisor or in a virtual machine (VM). None of them can trace activities, e.g. network processing, throughout the entire virtualized I/O stack. Many state-of-the-art works, like Mystery Machine [20], Draco [24], LogEnhancer [37], lprof [38], identify performance anomalies among machines based on distributed logs. However, analyzing massive logs, not only introduces non-negligible runtime overhead, but also cannot guarantee providing the information users need, e.g., statistics on tail latency. In addition, such out-of-band profiling needs additional effort to stitch the distributed logs [18, 33], and time drift among logs on different machines may also affect the accuracy.

In this paper, we propose *Time Capsule* (TC), a profiler to trace network latency at packet level in virtualized environments. TC timestamps packets at predefined tracepoints and embeds the timing information into the payloads of packets. Due to in-band tracing, TC is able to measure packet latency across different layers and protection boundaries. In addition, based on the position of tracepoints, TC can decompose and attribute network latency to various components of the virtualized network stack to locate the potential bottleneck. Further, TC incurs negligible overhead and requires

*APSys '16*,, August 04-05, 2016, Hong Kong, China.
Copyright © 2016 ACM 978-1-4503-4265-0/16/08. . . $15.00.
http://dx.doi.org/10.1145/2967360.2967376

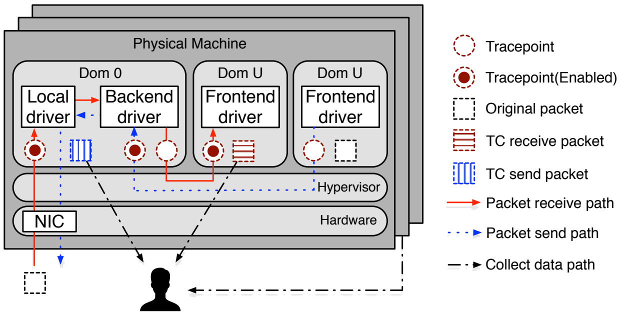

**Figure 1.** The architecture of Time Capsule.

no changes to the traced applications. We demonstrate that fine-grained tracing and latency decomposition enabled by TC shed light on the root causes of long tail network latency and help identify real performance bugs in Xen.

The rest of the paper is organized as follows. Section 2 and 3 describe the design, implementation, and optimizations of Time Capsule, respectively. Section 4 presents evaluation results. Section 5 and 6 discuss future and related work, respectively. Section 7 concludes this paper.

## 2. Design and Implementation

The challenges outlined in Section 1 motivate the following design goals of Time Capsule: (1) cross-boundary tracing; (2) fine-grained tracing; (3) low overhead; (4) application transparency. Figure 1 presents a high-level overview of how TC enables tracing for network send (Tx) and receive (Rx). TC places tracepoints throughout the virtualized network stack and timestamps packets at enabled tracepoints. The timing information is appended to the payload of the packet. For network receive, before the traced packet is copied to user space, TC restores the packet payload to its original size and dumps the tracing data to a kernel buffer, from where the tracing data can be copied to user space for offline analysis. For network send, trace dump happens before the packet is transmitted by the physical NIC. Compared to packet receive, we preserve the timestamp of the last tracepoint in the payload of a network send packet to support tracing across physical machines. Since tracepoints are placed in either the hypervisor, the guest kernel or the host kernel, TC is transparent to user applications. Next, we elaborate on the design and implementation of TC in a Xen environment.

***Clocksource*** To accurately attribute network latency to various processing stages across different protection domains, e.g., Dom0, DomU, and Xen, a reliable and cross-domain clocksource with high resolution is needed. The para-virtualized clocksource `xen` meets the requirements. In a Xen environment, the hypervisor, Dom0, and the guest OS all use the same clocksource `xen` for time measurement. Therefore, packet timestamping using the `xen` clocksource avoids time drift across different domains. Next, the clocksource `xen` is based on the Time Stamp Counter (TSC) on the processor and has nanosecond resolution. It is adequate for latency measurement at the microsecond granularity. A similar clocksource `kvm-clock` is also available in KVM. Furthermore, we enabled `constant_tsc` and `nonstop_tsc` of Intel processors, which can guarantee that TSC rate is not only synchronized across all sockets and cores, but also is not affected by power management on individual processors. As such, TSC ticks at the maximum CPU clock rate regardless of the actual CPU running speed. For cross-machine tracing, the clocks on physical nodes may inevitably tick at different rates due to different CPU speeds. Therefore, the relative difference between timestamps recorded on separate machines does not reflect the actual time passage. Figure 2 shows the relationship between the TSC readings on two machines with different CPU speeds. The slopes of the two lines represent the maximum CPU frequency on the respective machines. There exist two challenges in correlating the timestamps on separate machines. First, TSC readings are incremented at different rates (i.e., different slopes). Second, TSC registers are reset at boot time or when resuming from hibernation. The relative difference between TSC readings on two machines includes the absolute distance of these machines since last TSC reset. For example, as shown in Figure 2, the distance between TSC reset on two machines is denoted by $\alpha = |t^a_{reset} - t^b_{reset}|$, where $t^a_{reset}$ and $t^b_{reset}$ are the last TSC reset time of machine $a$ and $b$, respectively.

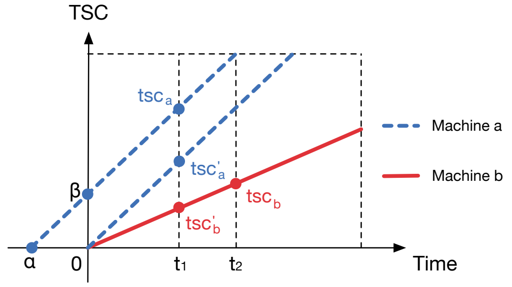

**Figure 2.** The relationship of TSCs on different physical machines. The constant_tsc flag allows TSC to tick at the processor's maximum rate regardless of the actual CPU speed. Line slope represents the maximum CPU frequency.

***Tracepoints*** are placed on the critical path of packet processing in the virtualized network stack. When a target packet passes through a predefined tracepoint, a timestamp based on local clocksource is appended to the packet payload. The time difference between two tracepoints measures how much time it spent in a particular processing stage. For example, two tracepoints can be placed at the backend in Dom0 and frontend in DomU to measure packet processing time in the hypervisor. As timestamps are taken sequentially at various tracepoints throughout the virtualized network stack, TC does not need to infer the causal relationship of the tracepoints (as Pivot Tracing does in [27]) and the timestamps in the packet payload have strict happened-before relations.

***Cross-machine tracing*** requires that the varying TSC rates and reset times on different machines be taken into account for accurate latency attribution. Specifically, timestamps recorded on separate machines should be calibrated to determine the latency due to network transmission between machines. We illustrate the TSC calibration process in Figure 2. Assume that a packet is sent from machine $a$ (denoted by the dotted blue line) at time $t_1$, which has a faster CPU and its TSC starts to tick earlier, and received at time $t_2$ on machine $b$ (denoted by the solid red line) with a slower CPU. Without TSC calibration, the difference $tsc_b - tsc_a$ can show negative transmission time. There are two ways to measure packet transmission time in the network based on the two timestamps $tsc_a$ and $tsc_b$ taken at the sender and receiver machines. First, the difference of TSC reset time $\alpha$ can be estimated as $|\frac{tsc^a_{sync}}{cpufreq_a} - \frac{tsc^b_{sync}}{cpufreq_b}|$, where $tsc^a_{sync}$ and $tsc^b_{sync}$ are the instantaneous TSC readings on the two machines at exactly the same time. This can be achieved through distributed clock synchronization algorithms which estimate the packet propagation time in a congestion-free network and adjust the two TSC readings. Once $\alpha$ is obtained, the absolute TSC difference $\beta$ is calculated as $\beta = \alpha \times cpufreq_a$. Then, the first calibration step is to derive $tsc'_a = tsc_a - \beta$ to remove the absolute TSC difference. As shown in Figure 2, $tsc'_a$ is the TSC reading of packet transmission at the sender if machine $a$ resets TSC at the same time as machine $b$. Further, the equivalent TSC reading at the receiver machine $b$ when the packet starts transmission is $tsc'_b = tsc'_a \times \frac{cpufreq_b}{cpufreq_a}$. Finally, the packet transmission time is the difference between the timestamps of packet send and receive on the receiver machine $b$: $t_2 - t_1 = \frac{tsc_b - tsc'_b}{cpufreq_b}$.

The first calibration method only requires the examination of one packet to measure packet transmission time but relies on an accurate estimation of $\alpha$. Since $\alpha$ is constant for all packet transmissions between two particular machines, an alternative is to estimate network condition based on the comparisons of multiple packet transmissions. Similar to [25], which compares packet transmission time with a reference value in a congestion-free environment to estimate network congestions, we can roughly measure packet transmission time as $\frac{tsc_a}{cpufreq_a} - \frac{tsc_b}{cpufreq_b}$ and use cross-packet comparison to identify abnormally long transmission time. However, this method only identifies relative transmission delays with respect to a reference transmission time, which is difficult to obtain in production datacenter network and may be variable due to packets being transmitted through different routes.

***Tracing payload*** To enable tracing latency across physical or virtual boundaries, TC adds an extra payload to a packet to store the timestamps of tracepoints. Upon receiving a packet at the physical NIC or copying a packet from user space to kernel space for sending, TC uses `__skb_put(skb, SIZE)` to allocate additional space in the original packet. The tracing information is removed from packet payload and dumped to a kernel buffer before a packet is copied to the application buffer in user space or sent out by the physical NIC. Figure 3 shows the structure of a TC-enabled packet. The tracing payload contains two types of data: the tracing raw data and the tracing metadata. The tracing raw data consists of 8-byte entries, each of which stores the timestamp of a tracepoint. Users can place plenty of tracepoints in the virtualized network stack based on their needs and choose which tracepoints to enable for a particular workload. The tracing metadata uses the annotation bits to indicate if the corresponding tracepoint is enabled or not (1 as enabled). Users define an event mask to specify the enabled tracepoints and initialize the tracing metadata. The `SIZE` of the tracing payload depends on the number of enabled tracepoints. For latency tracing across VMs on the same physical machine, packet is transferred between the hypervisor and domains through shared memory. The packet size is not limited by the maximum transmission units (MTUs). Thus, TC is able to allocate sufficient spaces in packet payload for tracing without affecting the number of packets communicated by the application workloads. For tracing across different physical machines, we dump all the timestamps before packets are sent out by the NIC but preserve the last timestamp recorded at the sender side (the sender side raw data in Figure 3) in the tracing payload. When the packet arrives at the receiver machine, new tracing data will be added after the sender side's last timestamp. As such, the tracing data pertaining to the same packet stored on multiple machines can be stitched together by the shared timestamp.

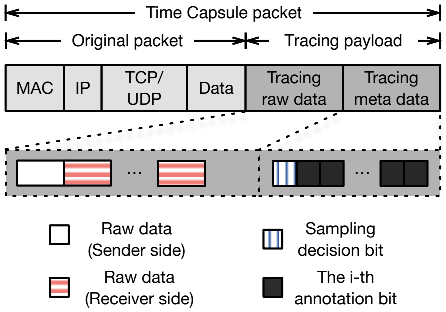

**Figure 3.** The structure of a Time Capsule packet.

## 3. Overhead and Optimizations

Despite the benefits, tracing at packet level can introduce considerable overhead to network applications. For highly optimized services in the cloud, such tracing overhead can significantly hurt performance. In this section, we discuss the sources of overhead and the corresponding optimizations in Time Capsule.

**Time measurement** It usually incurs various levels of cost to obtain timestamps in the kernel space. If it is not properly designed, fine-grained tracing can significantly degrade network performance, especially increasing packet tail latency. ***Optimization*** We compare the cost of different clocksource read functions available in various domains using a simple clock test tool [2] and adopt `native_read_tscp` to read from the `xen` clocksource at each tracepoint. Compared to

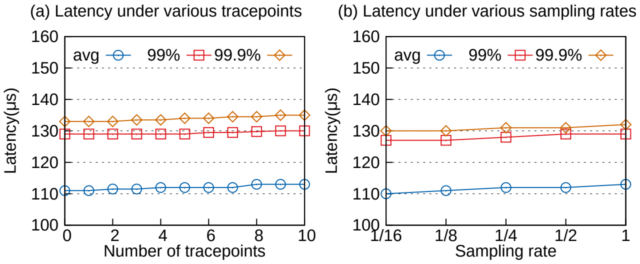

**Figure 4.** Time Capsule incurs negligible overhead to packet latency with various (a) number of tracepoints and (b) sampling rates. The sampling rate is set to 1/4 in (a) and the number of tracepoints is 10 in (b).

other time functions, which incur an overhead ranging from several microseconds to a few milliseconds, the function `native_read_tscp` adds only about 20 nanoseconds latency at a tracepoint. The overhead is negligible compared to tens and hundreds of microseconds latency in a typical network request.

**Trace collection** The dump of traces to storage is the most expensive operation in TC. If the completion of each packet triggers a disk write, the overhead will be prohibitively high. Further, the overhead grows as the intensity of network traffic increases.

***Optimization*** We adopt a ring buffer in the guest network stack (receiver side) and the NIC driver in the driver domain (sender side) to temporarily store the tracing data. Each time the tracing payload is removed from a packet, the tracing data is copied to the buffer. The latest trace sample overwrites the oldest data if the circular buffer is full. We use `mmap` to map the kernel buffer to user space `/proc` file system, from where the traces can be dumped to persistent storage. We design a user space trace collector that periodically dumps and clears collected traces. As trace collection happens infrequently and can be performed offline on another processor, it does not hurt network latency.

**Instrumentation cost** Too much instrumentation is one of the common issues in system monitoring, especially for latency-sensitive applications. As Google's tracing infrastructure Dapper [31] shows, over-sampling increases the average latency of web search by as much as 16.3%, while only inflicting a marginal 1.48% drop on throughput.

***Optimization*** We devise two optimizations to reduce instrumentation overhead. First, TC selectively traces packets from a targeted application. We use a combination of IP address and port number to identify the packets that should be traced. Other network packets skip the tracepoints and are processed normally. Second, as Figure 3 shows, tracing metadata in each packet allows TC to configure a flexible tracing rate (i.e., via the sampling decision bit) and to select a subset of tracepoints (i.e., via the annotation bit). The sampling decision bit enables tracing for a portion of packets in a network traffic based on a user-defined sampling rate. The annotation bit determines which tracepoint(s) should be enabled for a particular packet.

## 4. Evaluation

**Experimental Setup** Our experiments were performed on two PowerEdge T420 servers, connected with Gigabit Ethernet. Each server was equipped with two 6-core 1.90GHz Intel Xeon E5-2420 CPUs and 32GB memory. The host ran Xen 4.5 as the hypervisor and Linux 3.18.21 in the driver domain. The VMs had a single virtual CPU (vCPU) and 4GB memory, and ran Linux 3.18.21 as the guest OS.

### 4.1 Overhead Analysis

First, we analyze the overhead of Time Capsule on network latency. Figure 4 plots the average, $99^{th}$, and $99.9^{th}$ percentile latency of UDP packets using Sockperf [11]. We varied the number of tracepoints and the sampling rate. As shown in Figure 4, the number of tracepoints does not have much impact on packet latency, with no more than 1.5% latency increase (ten tracepoints) compared to that without TC (zero tracepoint). Similarly, the performance impact due to different sampling rates is insignificant. A sampling rate of 1 increased packet latency by 2% compared to that with a sampling rate of 1/16. As discussed in Section 3, TC's overhead mainly comes from timestamping the packet at tracepoints and manipulating packet payload, which takes tens of nanoseconds at each tracepoint. Given that typical network applications have latency requirements in the range of hundreds of microseconds to a few milliseconds, TC's overhead is negligible even with tens of tracepoints. In real systems, if enabling a large number of tracepoints for fine-grained tracing raises overhead concerns, lowering the packet sampling rate is likely to provide adequate trace data for network-intensive workloads. In summary, TC adds negligible overhead to network latency and overhead can be controlled by varying the number of tracepoints and the sampling rate.

### 4.2 Per Packet Latency

User-perceived latency is an important QoS metric. However, it is difficult to track individual user experiences in virtualized environments. When application-level per-request logging is not available, system-wide tracing can be effective in identifying performance issues of individual users. Next, we demonstrate that packet level tracing reveals problems that could be hidden in coarse-grained tracing.

We created a scenario in which users-perceived latency suffered sudden hike due to a short network burst from another application. We chose Sockperf as the application under test and used Netperf [9] to generate the interfering traffic. Figure 5 plots the per-second average latency and packet level latency of Sockperf. The burst arrived at the $1.5th$ second and left at the $6.5th$ second. We have three observations from Figure 5 (b): i) packet level latency accurately captured latency fluctuations during the burst; ii) packet level latency measurement timely reflected user-perceived latency immediately after the network spike left (at 6.5s); iii) most importantly, packet level tracing successfully captured a few

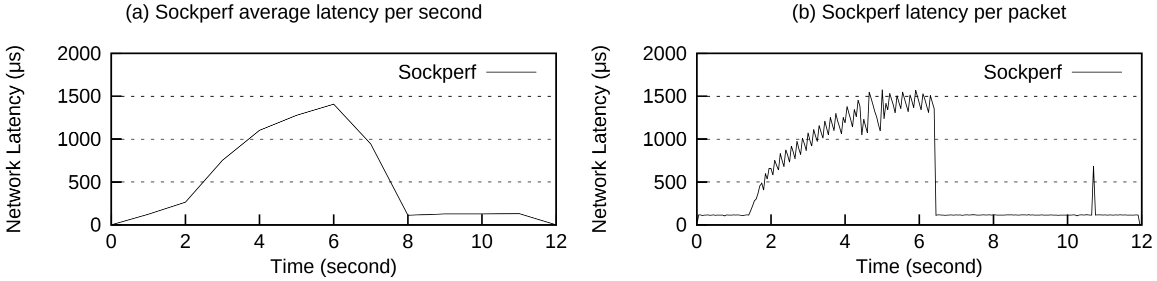

**Figure 5.** Compared to coarse-grained tracing, packet level tracing provides more information about user-perceived latency.

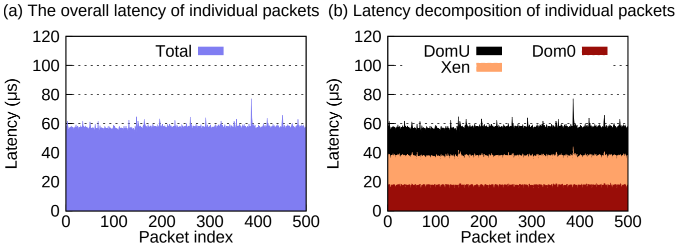

**Figure 6.** Latency decomposition when Sockperf runs alone in a VM.

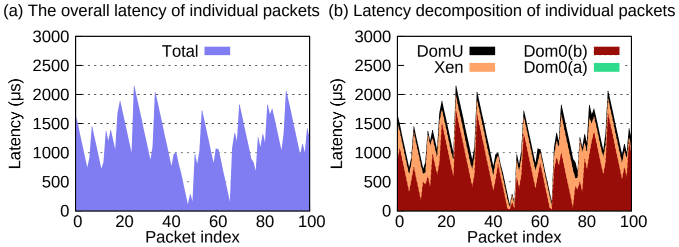

**Figure 7.** Latency decomposition when Sockperf is co-located with HPCbench in the same VM.

spikes in latency around the $11th$ second. The spike may be due to the backlogged packets during the traffic burst.

In contrast, per-second average latency (as shown in Figure 5 (a)) hid performance fluctuations, was not responsive to load changes, and missed the important latency issue after the bursty traffic left. Microbursts, the sudden and rapid traffic bursts in the network, can cause long tail latency or packet loss and may not be captured by coarse-grained monitoring. TC enables packet level latency tracing and can effectively identify slight performance changes. Next, we show that TC further decomposes the packet latency into time spent in different stages to help locate the root causes of long latency.

### 4.3 Latency Decomposition

A detailed breakdown of packet latency sheds light on which processing stage in the virtualized network stack contributes most to the overall latency and help identify abnormalities at certain stages. Figure 6 and Figure 7 show the latency and its breakdown of Sockperf under two scenarios: i) the VM hosting Sockperf alone; ii) the VM hosting Sockperf and HPCbench [8] simultaneously. In scenario ii, two separate clients sent Sockperf UDP requests and HPCbench UDP stream flows, respectively. While it is expected that the co-location of Sockperf and HPCbench in the same VM causes interference to Sockperf, we show that latency breakdown provides insight on how to mitigate the interference.

Figure 6 (a) and (b) show the receiver side latency for 500 packets and the latency breakdown when Sockperf ran alone in the VM. Without interference, the latency stabilized at about 60 $\mu s$ and the processing at the driver domain, the hypervisor, and the guest kernel contributed equally to the overall latency. In contrast, Sockperf latency degraded by up to 35x and became wildly unpredictable when co-running with throughput-intensive HPCbench workload, as shown in Figure 7 (a). Note that although interference also exists on physical machines when co-locating latency-sensitive workloads with throughput-intensive workloads, the performance degradation is not as drastic as that in virtualized environments. Latency breakdown suggests that most degradation was from prolonged processing in Dom0.

To pinpoint the root cause, we added two additional tracepoints to further break down packet processing in Dom0: i) packet processing in Dom0's network stack (denoted as *Dom0 (a)*) and ii) processing in the VM's backend NIC driver at Dom0 (denoted as *Dom0 (b)*). As shown in Figure 7 (b), most time in Dom0 was spent in the backend NIC driver. After an analysis of Dom0's backend driver code, we found that the excessively long latency was due to batching the memory copy between the backend and frontend drivers. The backend driver does not copy packets to a VM until the receive queue of the physical NIC is depleted. All received packets will be copied to a VM in a batch to amortize the cost of memory copy. This explains why workloads with bulk transfer degrade the performance of latency-sensitive applications. A large number of packets from throughput-intensive workloads fill up the receive queue in the backend driver, preventing the packets of latency-sensitive application from being transferred to the VM. The analysis based on latency decomposition suggests that limiting the batching size in the backend driver would alleviate the interference.

### 4.4 Case Studies

With packet level tracing and latency decomposition, TC helps associate the excessively long latency to certain processing stages in the virtualized network stack. In this section, we describe our discovery of multiple bugs in Xen's

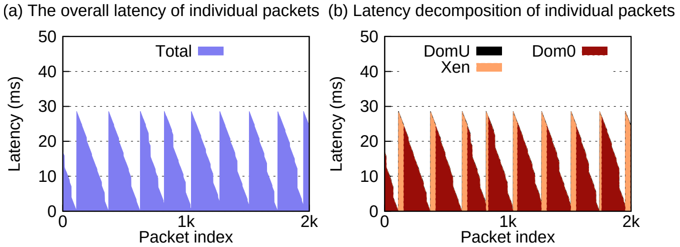

**Figure 8.** Bug-1 [3]: Xen mistakenly boosts the priority of CPU-intensive VMs, which invalidates the prioritization of I/O-bound VMs.

credit scheduler with the help of TC. These scheduler bugs cause long tail latency of I/O workloads. We reported the bugs to the Xen community and they were confirmed by engineers from Citrix [3–5]. Next, we explain how TC helped locate these bugs.

Xen's credit scheduler is designed to prioritize I/O-bound VMs while not compromising the overall system utilization and fairness. Thus, if an I/O-bound VM consumes less than its fair CPU share, it always has a higher priority than a CPU-bound VM and I/O performance should not be affected by co-located CPU-bound workloads. Figure 8 shows the performance of Sockperf when its host VM shared the same physical CPU with another CPU-bound VM. We observed that Sockperf latency degraded significantly due to the interference from the CPU-bound VM. For approximately every 250 packets, latency grew to as high as 30 $ms$ and started to descend until the next spike. Latency decomposition in Figure 8 (b) shows that latency spikes always started with long delays in Xen, which dominated the overall latency. This indicates that the latency spike started with packets being blocked in Xen and then the delay propagated to Dom0. The delay in Xen was close to 30 $ms$, which matched the length of the default time slice in the credit scheduler. These observations gave a hint that the long latency is correlated with the VM scheduler.

The analysis led to the discovery of the first bug in the credit scheduler: Xen mistakenly boosts the priority of a CPU-bound VM, thereby preventing an I/O-bound VM from being prioritized. If this happens, the I/O-bound VM needs to wait for a complete time slice (i.e., 30 $ms$) before the CPU-bound VM is descheduled by Xen due to the expiration of its time slice. The latency breakdown in Figure 8 (b) further shows that the first wave of packets in the latency spike were copied to the grant table in Xen but cannot be processed by the I/O-bound VM because the VM was not scheduled to run. Thus, the delay is attributed to the wait time in Xen. After the grant table was full but the I/O VM was not yet scheduled, new coming packets stayed in the backend driver of Dom0, which explained the propagation of delay to Dom0. After fixing this bug, both the average and tail latency were greatly improved.

Using the same methodology, we discovered another two bugs in the credit scheduler that also contributed to the long tail latency issue. The reasons behind excessively long tail latency are usually complicated. As packets with long latency are ephemeral and difficult to be captured by coarse-grained monitoring, finding the causes is even harder. Figure 9 shows the occurrence of abnormal latency in twenty thousands packets. Only 5% of the packets suffered long latency and their occurrence had no clear patterns. As the magnitude and occurrence of the abnormal latency were unpredictable, using TC traces alone was not enough to identify the bugs. As Figure 9 (b) shows similar patterns of the long latency – long delays in Xen followed by delays in Dom0, we separately instrumented Xen to report its scheduling events and performed side-by-side comparisons of the Xen trace and TC packet trace. We used event and packet timestamps to correlate the Xen scheduling events with abnormal packets. Considerable manual effort was needed to identify two additional bugs in the credit scheduler. Through the above examples, we have demonstrated that TC is quite useful to identify, localize and analyze latency issues, especially for the long tail latency.

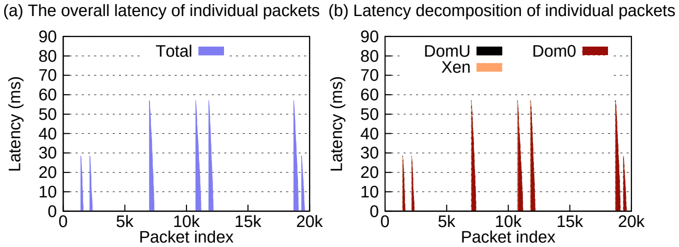

**Figure 9.** Bug-2 [4]: Xen does not timely activate I/O VMs that are deactivated due to long idling. Bug-3 [5]: I/O VMs' BOOST priority can be prematurely demoted.

## 5. Discussions and Future Work

***Packet size*** TC relies on the additional space in the packet payload to store tracing information. For small packets and the packets transferred between VMs on the same host, TC's tracing payload does not increase the number of packets needed by the original application. In the case of cross-machine tracing, an 8 byte space is needed for the last timestamp from the sender. It is possible that this will increase the number of MTUs transferred by the original application. When network is congested, the additional MTUs could be the source of overhead. For packets larger than the MTU, NIC that supports GSO/GRO features will split the large packets into separate MTUs at the sender side. TC only needs to append an 8-byte timestamp to the last MTU.

***Dynamic instrumentation*** Currently, we manually add tracepoints in TC, and Dom0 and DomU kernels need to be recompiled to use the new tracepoints. Dynamic instrumenting the Linux kernel is possible by using the extended Berkeley Packet Filter (eBPF). However, dynamic instrumentation cannot manipulate packet payload, thereby unable to cross the boundaries between Dom0, DomU and the hypervisor.

***Automated analysis*** Although TC provides detailed information about packet processing in virtualized systems, considerable manual effort is needed to identify the causes of unsatisfactory performance. TC can only correlate the stages on the critical path of I/O processing with the overall network performance. For performance issues due to other resource management schemes, such as VM scheduling and memory allocation, a causal analysis of TC trace and other system traces is necessary. Statistical approaches that find the correlation between the traces are promising directions towards automated identification of the root causes.

***Extending TC to disk I/Os*** TC leverages the commonly shared data structure `skb` to associate tracing information with individual packets. Virtualized disk I/O is as complex as virtualized network I/O and often suffers poor and unpredictable performance. However, it is more challenging to trace disk I/O across multiple block I/O stacks in virtualized systems. The challenges are the lack of a shared data structure between the protection domains to pass the tracing information and aggressive optimizations of disk I/O at different layers of the virtualized system.

## 6. Related Work

***Monitoring and tracing schemes*** There are in general two ways to monitor performance and trace program execution in complex systems. Non-intrusive tracing systems [14, 15, 20, 28–30, 35] leverage existing application logs and performance counters to diagnose performance issues and detect bugs. Annotation-based monitoring [16, 19, 22, 31] offers users more flexibility to selectively trace certain components in the monitored systems. However, both approaches still face the challenges of balancing the tradeoff between the effectiveness of tracing and its overhead. Dynamic instrumentation allows flexible logging and tracing to be installed dynamically. Pivot Tracing [27] leverages aspect-oriented programming to export variables for dynamic tracing and designs a query languages to selectively invoke user-defined tracepoints. Similarly, Time Capsule requires tracepoints to be manually inserted by administrator but supports a flexible selection of tracepoints and varying sampling rates. Different from Pivot Tracing on the distributed systems, TC is a low-level profiler that focuses on event tracing on the critical path of network I/O processing in virtualized systems.

***Tracing across boundaries*** Tools are widely used in complex systems to diagnose performance problems. However, many tools are limited within certain boundaries. For example, gperf [7] is an application-level tool to analyze UNIX program performance while Systemtap [12], DTrace [17], and Perf [10] are used to trace source code or collect performance data inside Linux kernel. Similarly, Xentrace and Xenalyze [6] can only be used to trace events in the Xen hypervisor. To address the limitations, many efforts have been proposed to trace beyond boundaries. Stardust [33] uses breadcrumb record associated with application requests to correlate events in separate components or computers. Whodunit [18] adopts synopsis, a compact representation of a transaction context, to profile transactions across distributed machines. Pivot tracing [27] uses a per-request container, *baggage*, to correlate logging information with a particular request context. There are also other works that infer the relationship between events in distributed environments [16, 18, 19, 22, 31, 33]. Inspired by [26], TC embeds the tracing information into the payload of network packets and dumps the traces before packets are received by applications. This design offers two advantages. First, events are naturally ordered in the payload, avoiding efforts to casually correlate logs from distributed machines. Second, in virtualized environments, it is not always possible to access tracing events in the privileged domain (e.g., the driver domain) from unprivileged domains (e.g., user VMs) due to security concerns. TC passes tracing information along with the I/O path, thus analysis only needs to be performed in user VMs.

***Latency measurement and analysis*** Much effort has been dedicated to analyzing factors that affect network latency. DARC [34] proposes runtime latency analysis to find main latency contributors. Li *et al.* [26] explore the potential causes for tail latency on multi-core machines. Similarly in virtualized environment, Soroban [32] studies the latency in VMs or containers by using machine learning and Xu *et al.* [36] introduce a host-centric solution for improving latency in the cloud. As Software Defined Network (SDN) and Network Function Virtualization (NFV) become popular in recent years, latency measurement and analysis play more important roles in modern networks. For example, relying on accurate timestamps provided by SoftNIC [23], DX [25] proposes a new congestion control scheme to reduce queueing delay in datacenters. TC is complementary to these approaches and can be used to evaluate their effectiveness.

## 7. Conclusion

Latency is a critical QoS factor for network applications. However, it is challenging to monitor latency in virtualized systems. This paper presents Time Capsule (TC), an in-band packet profiler that traces packet level latency across different boundaries in virtualized systems. TC incurs negligible overhead to network performance and requires no changes to applications. We demonstrated that fine-grained packet tracing and latency decomposition shed light on the latency problems and helped us identify three bugs in Xen's credit scheduler. Guided by TC, we are able to trace and analyze issues that cause long latency in virtualized systems.

## Acknowledgements

We are grateful to the anonymous reviewers and the paper shepherd, Dr. Rajesh Krishna Balan for their constructive comments. This research was supported in part by the U.S. National Science Foundation under grant CNS-1320122 and by Hong Kong Research Grants Council Collaborative Research Fund under grant C7036-15G.

## References

[1] *Amazon Elastic Compute Cloud(EC2)*. http://aws.amazon.com/ec2/.

[2] *Code to test various clocks under Linux*. https://github.com/btorpey/clocks.

[3] *VCPU is mistakenly woken up in Xen's credit scheduler*. http://lists.xenproject.org/archives/html/xen-devel/2015-10/msg02853.html, .

[4] *VCPU which is not put back to active VCPU list in time will cause unpredictable long tail latency*. http://lists.xenproject.org/archives/html/xen-devel/2016-05/msg01362.html, .

[5] *The BOOST priority might be changed to UNDER before the preemption happens*. http://lists.xenproject.org/archives/html/xen-devel/2016-05/msg01362.html, .

[6] *Tracing with Xentrace and Xenalyze*. https://blog.xenproject.org/2012/09/27/tracing-with-xentrace-and-xenalyze/, .

[7] *GNU gperf*. https://sourceware.org/binutils/docs/gprof/.

[8] *HPCbench*. http://hpcbench.sourceforge.net/.

[9] *Netperf*. http://www.netperf.org/.

[10] *Linux Perf*. https://perf.wiki.kernel.org/.

[11] Sockperf. https://github.com/Mellanox/sockperf.

[12] *SystemTap*. https://sourceware.org/systemtap/.

[13] *Xentrace*. https://blog.xenproject.org/2012/09/27/tracing-with-xentrace-and-xenalyze/.

[14] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of ACM SIGOPS Operating Systems Review (SOSP)*, 2003.

[15] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of ACM SIGCOMM Computer Communication Review (SIGCOMM)*, 2007.

[16] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[17] B. Cantrill, M. W. Shapiro, A. H. Leventhal, et al. Dynamic instrumentation of production systems. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*, 2004.

[18] A. Chanda, A. L. Cox, and W. Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. In *Proceedings of European Conference on Computer Systems(EuroSys)*, 2007.

[19] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of Dependable Systems and Networks (DSN)*, 2002.

[20] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[21] Ú. Erlingsson, M. Peinado, S. Peter, M. Budiu, and G. Mainar-Ruiz. Fay: extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS)*, 2012.

[22] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2007.

[23] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. Softnic: A software nic to augment hardware. In *Technical Report UCB/EECS-2015-155*. 2015.

[24] S. P. Kavulya, S. Daniels, K. Joshi, M. Hiltunen, R. Gandhi, and P. Narasimhan. Draco: Statistical diagnosis of chronic problems in large distributed systems. In *Proceedings of Dependable Systems and Networks (DSN)*, 2012.

[25] C. Lee, C. Park, K. Jang, S. Moon, and D. Han. Accurate latency-based congestion feedback for datacenters. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*, 2015.

[26] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2014.

[27] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[28] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[29] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. Wap5: black-box performance debugging for wide-area systems. In *Proceedings of the 15th international conference on World Wide Web (WWW)*, 2006.

[30] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[31] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. *Google research*, 2010.

[32] J. Snee, L. Carata, O. R. Chick, R. Sohan, R. M. Faragher, A. Rice, and A. Hopper. Soroban: attributing latency in virtualized environments. In *Proceedings of the 7th USENIX Conference on Hot Topics in Cloud Computing(HotCloud)*, 2015.

[33] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: tracking activity in a distributed storage system. In *Proceedings of the ACM SIGMETRICS Performance Evaluation Review (SIGMETRICS)*, 2006.

[34] A. Traeger, I. Deras, and E. Zadok. Darc: Dynamic analysis of root causes of latency distributions. In *Proceedings*

*of the ACM SIGMETRICS Performance Evaluation Review (SIGMETRICS)*, 2008.

[35] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)*, 2009.

[36] Y. Xu, M. Bailey, B. Noble, and F. Jahanian. Small is better: Avoiding latency traps in virtualized data centers. In *Proceedings of the 4th annual Symposium on Cloud Computing (SoCC)*, 2013.

[37] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)*, 2012.

[38] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.