**Performance Analysis of Virtual Environments**

**Introduction**

Virtual machines are becoming an integral part of both industrial data center facilities and large academic experimentation environments. It’s important to measure the effects of virtualization and identify performance bottlenecks in the virtual setups. This project aspires to provide tools to perform fine-grained analysis of resource utilization, virtualization overheads, and performance bottlenecks, in a virtualized software stack. We implement our performance analysis framework for the Xen virtual machine monitor, which is a full feature, open source hypervisor, and a de-facto industry standard for industrial data-centers.

Our project consists of two steps. First, we instrument the Xen hypervisor to collect a variety of fine-grained performance events, for example, VM scheduling, exception, page faults, interrupts, and so on. To answer questions about performance of virtual device drivers, we extend our tracing infrastructure to collect events from the device driver VMs, which provide these services.

Second, we implement an off-line trace analysis framework, which allows development of various performance analysis algorithms in a convenient, reusable and composable way. Each analysis algorithm consists of a collection of event handlers, which observe specific performance events. The framework allows the development of standalone tools as wells as compose new analyses on the fly using reusable components

*Fig 1: Xen and Xentrace Architecture*

XEN

Hardware

. . . .

Dom 1

Dom N

Xentrace

Domain events

Domain events

Hypervisor events

Reader

Analyses

HDD

Xentrace logs

Dom 0

1. **Xen Overview**

The Xen hypervisor allows multiple guest operating systems (Domains) to share hardware resources of a single physical host, as shown in Fig 1. Domain 0 is the administrative domain with elevated privileges and also hosts the backend device drivers in this case.

1. **Data Collection**

Xentrace is a lightweight tracing utility that ships with Xen, which collects hypervisor level events. Xentrace consists of 3 parts: tracing macros, fast communication channel and user level daemon application. Xentrace is typically run from Dom 0(control VM) and a daemon periodically dumps these buffered events to the disk (Fig 1). The amount of data collected by Xentrace can be enormous. Although this raw data can be quite useful during debugging, it does a poor job of providing a high level view of performance problems. For example, data collected during a disk intensive operation for 1 minute easily exceeds 700 MB or about 20 million lines in the log. To extract meaningful information from such a log by manually inspecting them, can be both time consuming and prone to errors.

Due to this enormous information collected by Xentrace, these trace logs are chosen to be our source of performance data. Trace macros are trivially easy to write. So to collect additional performance events not provided by Xentrace, one could easily insert appropriate trace macros inside the Xen or domain (Linux) source code. Some examples of the additional trace macros that were added are block device driver queue events, transition to and from Xen among others.

1. **Analysis Framework**

The framework was initially designed to have 2 main components: *Reader* and *Analyses* (Fig 1). The Reader component parses the binary Xentrace logs and neatly constructs C style data structures that are eventually passed to the Analyses component. The Analyses consists of a bunch of handlers for different types of events. These handlers are the only things that need to be written by the developer to build customized analysis tool. Some of the analyses implemented for this project are discussed in section 4.

* 1. **Reader**

Given that these logs are in binary format parsing them should be trivial, but a few caveats make it slightly more complicated.

Firstly, these logs are not *completely* recorded in temporal order. Let me elaborate. As shown in Fig 1 and briefly discussed in Section 2, Xentrace daemon periodically dumps the recorded trace buffers to the disk. These buffers are CPU specific i.e. every physical CPU has one buffer allocated to it. So if you have different guest domains executing on the same CPU, then both these domains have their traces collected in the same buffer. Consequently, all these buffers dumped in a single log file are not exactly sequential.

A trivial solution to this problem is to sort the data by timestamps and then parse the log, which was our initial approach. But over time, as we began running longer experiments, log files grew larger and sometimes larger than the physical memory the test machine possessed. In-memory sorting was no longer feasible. The next best approach was to use an external sorting technique like *d*-way Merge sort. But as mentioned in the last paragraph, the data recorded is not completely unordered. Data recorded inside each buffer is sorted. These buffers can then be thought of as *d*-sorted lists. So instead of implementing a generic external merge sort we used these buffers as the input, by tracking them individually in the log, and just implemented the final merge step. Thus we stream through these buffers returning the earliest occurring event.

The second caveat is the *lost\_records* event. This event is recorded in the logs whenever Xentrace fails to collect events for some period of time, usually due to the buffers being full. This missing data can interfere with our analysis, especially the ones that are time sensitive. We tried to minimize this problem by increasing the size of the buffers, collecting only a subset of events when possible and even fixed a bug in Xentrace, which prevented the complete collection of the logs. In spite of this, the problem did not completely disappear and usually crops up during periods of intense activity. To solve this problem, we decided to treat *lost\_records* as just another event in our analysis and notify its occurrence and the time lost to other event handlers. Based on this data, the tools can either choose to ignore it or restart the analysis.

* 1. **Analyses**

Each tool has an *Analyses* component that is made up of a list of handlers for events that are required to conduct the analysis. At the beginning of execution, these event handlers register their respective events with the *Reader.* The *Reader* tries to match the parsed event with the ones registered and passes it along to the matched event handler.

Every event handler consists of 3 main methods: *initializer* (initializes data), *handler* (actual analysis logic) and *finalize* (optionally aggregate the collected data, free malloc’d memory etc). These are the only 3 methods that the user needs to write for a new analysis tool. Examples of analysis tools built are discussed below in Section 4.

1. **Analysis Algorithms**

The motivation for many of the analysis tools was to quantify phenomena usually observed only in virtual setups. For example, we started with the most obvious one, CPU utilization. We soon realized that monitoring only the physical CPU usage is not very informative since domains can run on multiple CPUs or vice versa. Hence we also began monitoring CPU and VCPU utilization for each domain. Even with this granularity the picture was incomplete. Since each domain has to be scheduled by the hypervisor for execution, it’s possible that a delay in scheduling can starve a domain and hence would not be visible on CPU utilization metric. CPU utilization can also sometimes wrongly attribute resource usage. Since Xen uses a split device driver model, some privileged operations can execute either in the device driver domain or the hypervisor on behalf of a guest VM. As you can see reasoning about performance in a virtual setup is not exactly straightforward. Many times the developer or user has to follow the data or his instinct. Our framework allows the user to whip up a new tool in little to no time that suits his purposes.

Some of the analysis tools implemented for this project are,

* *CPU Utilization*: Apart from the Physical CPU core usage, this analysis also shows CPU usage by each guest VM and their respective Virtual CPU (VCPU) utilization. Some scenarios where such granularity is necessary are:
  + Check if hypervisor configuration adheres to Service Level Agreements (SLA) between hosting providers and clients.
  + Detecting unbalanced mapping between physical CPUs and VMs.
* *CPU Scheduling Latency*: Most operating systems running in non-virtualized environments have uninterrupted access to the physical CPUs they are running on. But when running inside Xen, access to CPU is multiplexed, resulting in situations where the guest domains may have to wait on the CPU. Such a situation can arise due to overscheduling of VCPUs or a physical CPU being busy, consequently delaying VM context switches. This analysis measures how much time a VM had to wait to get scheduled since the request to context switch was sent out.
* *Time in Hypervisor*: CPU utilization is not always reliable when trying to identify performance problems. For example, during an I/O intensive operation, a VM may spend much of its time being idle while the hypervisor does the work on its behalf on a different physical CPU. Hence this may give a wrong picture to someone who just relies on CPU usage data of a guest domain, as it doesn’t accurately represent the time spent in hypervisor. It can also help identify bottlenecks in the execution (Ex: high latency I/O).
* *Disk I/O*: Xen has a split device driver model where part of the driver resides in guest domain (frontend) and rest (backend) of it in the device driver domain usually the control domain, Dom 0. Hence a block device request from a guest VM navigates through many different queues like the VM elevator queue, Xen Shared ring buffers, queues inside Dom 0 etc., before the request actually reaches the HDD. The response also traverses the same path.
  + To identify the overheads of split device driver model and identify bottlenecks in the disk I/O pipeline, we built analysis to monitor the state of most of these queues and ring buffers to see how long they were blocked (cannot accept new requests) and unblocked (open to new requests). The intuition was that latency in the I/O path could be pinpointed if any of the queues stays blocked for a long time.
  + Xen provides an event channel mechanism that is used by guest domains and Dom 0 to notify each other when there is a request or a response on the shared ring. Any delay in processing this notification can also add to the latency, even when queues are unblocked.

1. **Composibility**

Although our framework made it easier to create new tools, building and using many of them gave us insight into what it lacks and what could be improved.

* 1. **Motivation**
* Firstly, many handlers shared lot of similar logic with the exception being what type of events they each handled. E.x: Both *CPU Scheduling Latency* and *Time in Hypervisor* ultimately take the cumulative sum of differences of two event timestamps. If ever we need to do an analysis, which requires similar logic, we would either have to create a new tool or modify an existing one.
* As indicated in Section 4, to make reasonable deductions about performance we probably have to consider results from multiple tools and sometimes even compare them during a specific context.

E.x: Looking at both *CPU Utilization* and *Time in Hypervisor* is logical, but both these tools give a good overall picture. If we needed more context, more granularity, like say we would like to see how much time was spent in the hypervisor when CPU utilization was greater than say 70%, then there is no straightforward way achieve this with the framework we had at this point.

As you can tell, what we needed was a way to *compose* new analysis using reusable components or existing tools. To meet this goal we came up with a simple language, based on a streaming language [1], which allowed us to create different and complex configurations of analysis.

A simplified grammar of the syntax is shown below,