Performance Analysis of Virtual Environments

Nikhil Vidyadhar Mishrikoti
(nikmys@cs.utah.edu)

Introduction

Virtual machines are becoming an integral part of both industrial data center facilities and large academic experimentation environments. It is 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

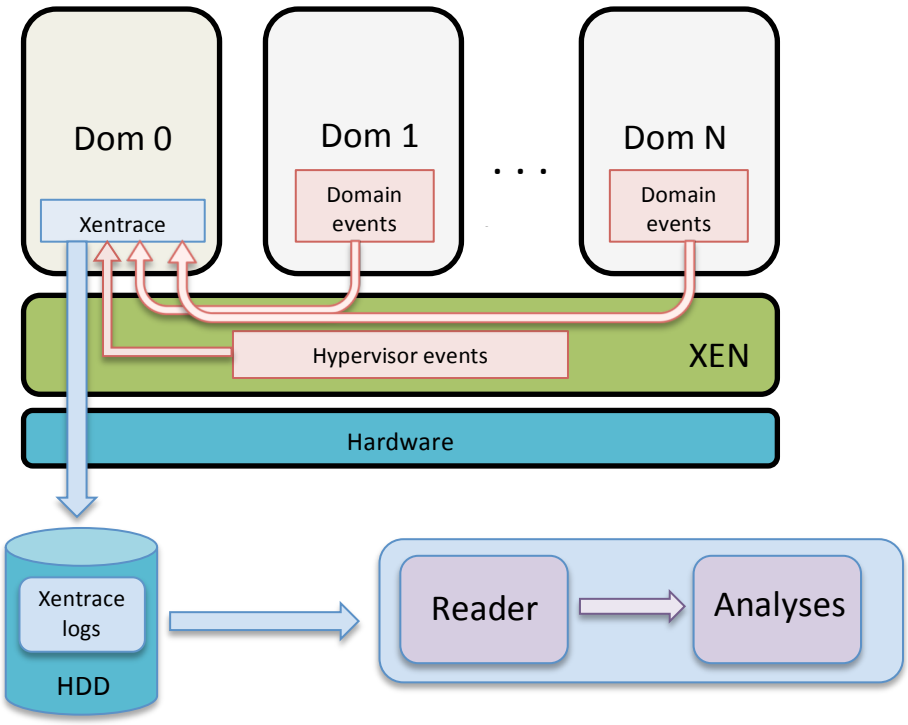

Fig 1: Xen and Xentrace Architecture

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.

2. 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 a user-level daemon application for collecting the log events. Xentrace is typically run from Dom 0(control VM) and the 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, but not all events of interest to us are included in the default Xentrace implementation – these include block device driver queue events, hypervisor entry and exit events and others. As the first stage of our work, we added additional events to Xentrace to collect these data points.

3. Analysis Framework

The framework was 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.

3.1. 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. As shown in Fig 1 and briefly discussed in Section 2, the 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 there are 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 that 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.

3.2. 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). The *handler* is the most important of these methods since it does the actual processing. Many times the *handler* needs to maintain a global state across every call of itself. Ex. If you are keeping a count of the events recorded in the log. The *initializer* method is usually used to allocate and initialize such a global state, and the *finalize* method for operating on this accumulated data and finally destroying this state. These are the only 3 methods that the user needs to write for a new analysis tool.

This is a reasonably good and simple model to construct new tools as the user can just reuse the *Reader* component, which takes care of parsing the logs, picking the earliest event and also filters events not registered by the tool. Allowing the user to focus his attention solely on handling events of interest. Since the tool can register to handle as many events as possible, analysis can be very computationally expensive. Running it alongside Xentrace can interfere with the performance data being collected. Consequently this analysis is run offline when the logs have been finished collecting. Examples of analysis tools built are discussed below in Section 4.

4. 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. There are two views of CPU utilization that one needs to consider. One from inside the guest VM for which there already exist tools and the other of physical CPUs from outside as Xen sees them. We began with physical CPUs only but 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. 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 create a new tool in little to no time that suits his purposes.

An operator would typically follow the USE (**U**tilization **S**aturation **E**rrors) methodology [1] to look for performance bottlenecks, before diving deeper. Since majority of the problems usually are found in the USE category, it made sense for us to build default tools that focus on these areas.

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 frontend request queues, Xen Shared ring buffers, backend queues inside Dom 0 etc., before the request actually reaches the HDD. The response also traverses the same path.

    - We built an analysis tool 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 if any of the queues is blocked for a long time then it blocks the entire pipeline.
    - I/O requests are typically sent on the shared ring buffer between Dom 0 and guest VM. Xen provides an event channel mechanism that is used by guest domains and Dom 0 to notify each other when there is an I/O request or a response on this shared ring. Any delay in processing this notification can also add to the latency.

    We ran this tool over two different configurations of block devices namely disk I/O with buffer cache and one without buffer cache.

    - In case of un-buffered disk I/O, we saw that all the queues we were monitoring stayed blocked over 99% of the time. Since disk access is so slow, the overhead of virtualization is negligible.
    - In case of buffered disk I/O, the frontend request queue in guest VM is blocked for less than 50% of the time but the remaining queues stay blocked for ~95% of the time. Due to the presence of the buffer cache, most disk write requests are consumed at the frontend device driver but wait to be flushed in the backend.

    As mentioned above we also measured latency of event notifications on the shared ring and noticed some interesting phenomena for buffered disk I/O. We define *wait time* as the time difference between the point a notification was sent from guest to Dom 0 or vice versa and when that notification was actually serviced. The *wait time* between Backend Shared Ring Response Queue and Frontend Shared Ring Response Queue is about two orders of magnitude larger than the *wait time* between Frontend Shared Ring Request Queue and Backend Shared Ring Request Queue.

These tools and results show how fine grained of an analysis we can conduct.

5. 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.

```
LOST RECORDS
=====================================================================
Total lost_record occurences = 0
Total lost time  =      0.000 (ms)

QUEUE TIMES
=====================================================================
Queue BLOCKED:   Unable to add new requests to queue or queue empty.
Queue UNBLOCKED: Can enqueue new incoming requests.

Front Request Queue Unblocked          :  43.209 (ms) ; Blocked : 26357.844 (ms)

Back Request Queue Unblocked           : 304.728 (ms) ; Blocked : 26096.322 (ms)

Front Shared Ring Resp Queue Unblocked :  59.295 (ms) ; Blocked : 26341.666 (ms)

QUEUE WAIT TIMES
=====================================================================

Back Request Queue Wait Time  :          38.301 (ms)

Back Response Queue Wait Time :          93.136 (ms)
```

Fig 2: Example output of Disk I/O tool for un-buffered disk

5.1. Motivation

- Firstly, many handlers share a lot of similar logic with the exception being what type of events they each handle. 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 see, it would be convenient if we could combine multiple analyses algorithms in different contexts. 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, which allowed us to create different and complex configurations of analysis. An operator could try to combine a variety of existing performance analysis to quickly identify performance bottlenecks. A deeper exploration might require development of additional analysis functions, as described in the previous sections.

5.2. Language implementation

In the core of our language is a runtime, which implements a simple streaming language. We base this runtime on the ideas of the general calculus for streaming languages [2]. We call the complete composed analysis as the *pipeline* and each discrete analysis a *stage*. The runtime implements the pipeline construction operators (*pipe*, *split*, *or*, *join*) that connect the stages, stage execution and pipeline execution.

To construct the pipeline, we rely on the PEG [3] parser generator Vembyr [4]. Vembyr generates an input language parser from the PEG grammar definition. The resulting parser is used to parse the input string and convert it in a general AST representation. We then use the AST to construct the pipeline by invoking the functions from the core runtime. The pipeline is then executed on the collected trace log.

5.2.1. Runtime

Before we proceed let's take a brief look at the function of the pipeline operators and then illustrate their use with an example. Basically, the function of each operator is to pass the events returned from preceding stages to the successive stages. If the previous stage returns an INVALID type event, then we halt execution of the pipeline and recursively return to the previous stage.

- *Pipe* (|) : Connects two single stages. Passes the returned event from the previous stage to the next stage.
- *Or* (**or**) : Connects a single stage to multiple stages. Runs all the stages it connects to in round robin fashion until one of the stage is successful (valid event returned).
- *Split* (**+**) : Connects a single stage to multiple stages. Runs all the stages it connects to in round robin fashion.
- *Join* : Connects multiple stages to a single stage. Depending on the type of join, we either wait for a single connected stage to pass a valid event (JOIN_OR) or wait for all the connected stages to pass a successful event (JOIN_SPLIT).

Let's take an example of a composable analysis. Suppose we want to know the average wait times between two events on a particular guest VM, the pipeline would look like something shown below.

```
vm(dom_id) | event_id(ev_block) + event_id(ev_unblock) | wait_time() | average()
```

Fig 3: Executable Pipe constructed using our syntax

```
s1 = create_stage(vm, dom_id);
s2 = create_stage(event_id, ev_block);
s3 = create_stage(event_id, ev_unblock);
s4 = create_stage(wait_time, NULL);
s5 = create_stage(average, NULL);

split(s1, s2);
split(s1, s3);
join(s2, s4, JOIN_SPLIT);
join(s3, s4, JOIN_SPLIT);
pipe(s4, s5);

while(!feof(fp))
{
      parse_next_event(&ev);
      execute_pipe(s1, ev);
}
```

Fig 4: Executable Pipe constructed programmatically

In the above example we have a total of 5 stages: *vm*, *event_id* (twice), *wait_time* and *average*. Each stage is connected using one of the four operators mentioned earlier. Each stage accepts an event, which is either passed to it from the parser or from a previous stage. Thus every event in the log file flows into the

pipe and upon processing by a stage, it is either passed to the next stage unmodified or a new event can be passed depending on the functionality of the stage it was passed into.

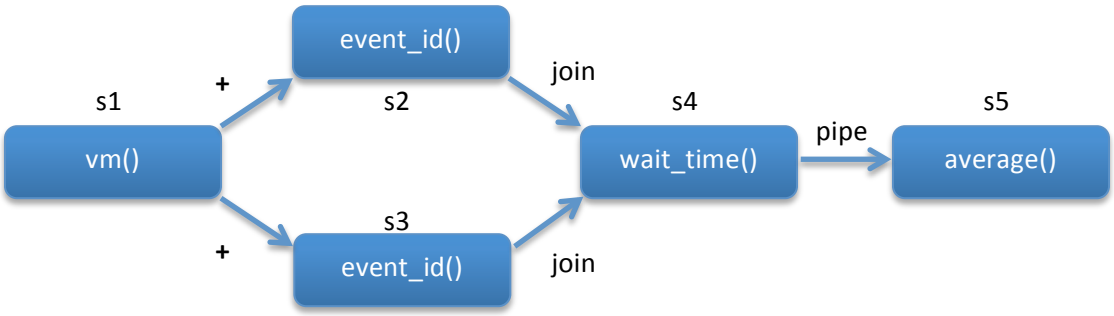

Fig 5: Executable Pipe shown as a tree.

In this example, the first stage, s1, acts as a filter and checks if the event passed to it belongs the domain/VM id setup for that stage. If there's a match, then that event is passed unmodified to branched stage s2 and s3 both. Join is slightly more complicated as depending on which branches we are joining we may either decide to wait until all the branches produce a valid event (SPLIT) or until any branch produces a valid event (OR). In our case, stage s4 (wait_time) requires two events to calculate the difference. So we proceed with the execution of the pipeline at the moment when we have valid input from both stages s2 and s3 at some point. Stage s5 is piped to s4, so execution is quite straightforward. But s4 sends a new event to s5, which contains the wait time data.

6. Conclusion

This project set out to make it easier for developers or Xen users to investigate varied performance problems in a virtual environment. This investigative process is far from straightforward and thus we strove to build a framework that enables the user to build tools that analyze the performance data with as much ease and as fine grained as possible. As the project progressed, we realized we could make constructing the tools even more flexible by giving the user the ability to compose new analysis tools using reusable components.

This material is based upon work supported by the National Science Foundation under Grant No. 1059440. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

7. References

[1] Thinking Methodically about Performance – Brendan Gregg, ACMQueue Dec 2012
[2] A Universal Calculus for Stream Processing Languages – Robert Soule et al, ESOP 2010
[3] Parsing expression grammars: a recognition-based syntactic foundation – Bryan Ford, POPL '04
[4] Vembyr – Multi-Language PEG Parser Generator [http://code.google.com/p/vembyr/]