Hypervisor Visualization

Cathy Cao, Christopher Mackie, Daniel Li, Nick Anderson, Wei Lin
{cathycao, mackic, dli123, nja4, veilam} @ cs.washington.edu

ABSTRACT

In this paper, we present our hypervisor visualization capstone. We first introduce the motivation and overall design of this project. We provide some background information to understand our work. We then walk through our instrumentation of the hypervisor and guest operating system and the control flow for the data collection. We finally discuss our visualization results and potential extensions to our project. Throughout, we describe the interface between the layers of our system.

1. INTRODUCTION

Hypervisors provide the means of running multiple OSs managed by a single entity. As platform-as-a-service and infrastructure-as-a-service offerings grow as the backbone of modern applications, providing OS-level isolation for customers running within a single physical machine becomes more and more important. A common problem encountered by these cloud providers is managing resources allocated to each virtualized machine and displaying the resources consumed to customers to justify billing. This is a standard feature of any modern cloud provider. We endeavor to visualize components of lvisor, and the OSs running within it, to solve this problem. In addition, we provide an interactive web console, similar to the console provided by DigitalOcean [2].

2. BACKGROUND

We use lvisor as the underlying hypervisor that runs a single guest operating system. The guest operating system we run is xv6, and because lvisor currently only supports one guest operating system, the extended page table, EPT, of lvisor uses identity memory mapping. We visualize the hypervisor and guest operating system in three ways: CPU time, memory usage, and disk usage. We instrument xv6 by adding hypercalls to collect relevant data in each category. Furthermore, we set conditional VM exit flags to break on other events of interest, such as IO. After logging, we parse the logs to generate a visualization of the log contents.

3. DESIGN

The goal of our visualizations is to determine the memory, CPU, and disk usage of a guest operating system, specifically xv6, running on the hypervisor lvisor. These three fundamental areas reflect the overall performance of a system, and visualizing their usage provides key insight to not only the underlying system's efficiency but also how other processes interact with the system.

In order to gather data for these areas, we utilize the vmcall instruction to trap into the hypervisor whenever related instructions and system calls are executed in the guest operating system. In this way, we log all the information we need in a temporary file, which is later pipelined to a Flask web server for visualizations using D3. We use Jinja2 to render templates and pass objects through to D3.

However, our temporary file based solution failed to capture output from the running xv6 operating system. We chose not to replace every occurrence of print to the console in xv6 with an explicit vmcall, but rather chose to break on writes to the COM port assigned for console output. First, we changed, in xv6, the console output to output to COM2, then created an IO bitmap wherein we marked COM2 writes to cause a VM exit. An alternative and similar design, with the fewest modifications to xv6, would rather leave console inputs and outputs both on COM1, then use the VM exit data structure returned to determine the instruction that caused the exit, re-executing console input and just logging console output.

To visualize our data in memory usage, disk usage, and CPU time, we use a grid map for memory and disk usage and a donut chart for CPU time. We chose a grid map for memory usage to show the individual pages in memory and highlight which pages are being allocated and deallocated by the guest OS. We also use a grid map for disk usage to color-code regions on disk and emphasize the disk blocks that are being written to in different regions during some program execution in the guest OS. A donut chart helps us visualize CPU time in lvisor partitioned by the set of VMexit reasons we encounter since booting lvisor and xv6.

4. CHALLENGES

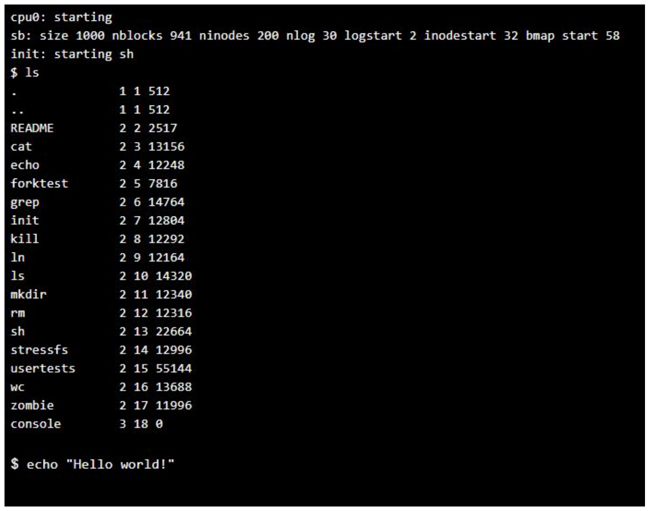
In designing and implementing our project, we encountered and overcame several challenges.

We initially planned to visualize the memory mapping between the guest OS and the hypervisor. However, in the case of lvisor, the hypervisor and the guest OS have identical memory mapping. Therefore, we needed to think of a way to visualize memory usage that is informative and interesting. We considered and now use a grid map visualization that shows the distinctions between memory allocated for lvisor and for xv6 and also whether pages in memory are actually used or free.

For CPU time usage, we recognized that the additional hypercalls lead to some overhead in program execution due to frequent VM exits and entries. However, this timing data can be informative to see the differences in CPU usage for each hypercall reason. We also needed to find an appropriate refresh time for the CPU time graph such that the graph is not outdated but also not refreshed too frequently that data logging and parsing cannot keep up. We decided to update the visualizations with new data pipelined from the log every second. We record the VMexit reason for a given data value and group data based on VMexit reasons to visualize the overall CPU time spent in lvisor between a VMexit and VMentry.

With our temporary file-based technique for capturing output, we have the challenge of piping user input into a Python subprocess, which is the current means of capturing output. Another design trade off is having a monolithic Flask app for simplicity versus having a running database that is updated with a log parsing function that we can draw from instead. To interface with Flask, we chose the simpler output file-based approach rather than a networking based approach due to the massive complexity involved without any functional gain. However, we investigated the use of lwip and the creation of an e1000 driver such that instead of vmcall instructions, we could instead serve data directly through TCP [3]. Because of the difficulties in setting up networking, we continue to use the temporary log file to record our data and we parse the logs to categorically gather data that is then piped to D3.

For the web console, the main challenges were ergonomic. After reading the character output to the console by examining register values after breaking on IO, we directly logged the output characters in lvisor, one at a time, as they were output on COM2. Then, we created the input layer, wherein we wrote to the “make qemu” subprocess’ stdin through another Flask route. Combining the two, we created the UI for printing out the xv6 console output and accepting input. After discussing with potential users, we opted for a design in Figure 1 that closely emulates what a user of xv6 would interact with.



```
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2517
cat            2 3 13156
echo           2 4 12248
forktest       2 5 7816
grep           2 6 14764
init           2 7 12804
kill           2 8 12292
ln             2 9 12164
ls             2 10 14320
mkdir          2 11 12340
rm             2 12 12316
sh             2 13 22664
stressfs       2 14 12996
usertests      2 15 55144
wc             2 16 13688
zombie         2 17 11996
console        3 18 0

$ echo "Hello world!"
```

Figure 1: Interactive Web Console UI

5. DATA COLLECTION

The mechanism for data collection is VMCALL. VMCALL is an instruction executed by the guest OS to voluntarily hand control over to the hypervisor, with the argument stored in %edi and VMCALL number in %eax. Lvisor then executes the corresponding function with the VMCALL number and log datas into a temporary local file. This can also be referred to as a hypercall.

For CPU usage, we are interested in how much time is spent inside lvisor and the reason behind it. Therefore, whenever there is a VMEXIT, we first log the current time of the VMEXIT and then the reason of the VMEXIT, which could be a VMCALL, a disk allocation, a cr3 access etc.

As for memory allocation, we want to know the total number of pages that are occupied by xv6 and lvisor. Inside xv6, whenever the OS needs to allocate or free memory, it calls kalloc or kfree respectively, both of which return the base address of the page allocated or freed. Therefore, inside those two functions, before they return the address, we trigger a VMCALL and pass in the base address. Inside the VMCALL, lvisor simply prints the base address and indicates whether it is allocated or freed.

Lastly, for disk usage, we want to log disk writes. We had to be careful to only consider the disk itself, and disregard the block cache, as this information is implicitly visualized in our memory visualization. True disk writes only occur in xv6 when the block cache is flushed. This occurs in iderw, which determines if a block is dirty, and writes it to disk if it is. Immediately after this write we use a hypercall to pass lvisor the number of the block which was flushed.

To transform the data into the correct format, we utilize the python web server to parse the temporary file into the correct data structures for transmitting to the web client. Every time we get pinged for data, we re-parse the updates contents of the temporary data file.

A problem we ran into when piping data to D3 was that it quickly started to lag as the dataset size grew. Initially, we would just re-render the entire dataset on the client side every couple seconds, but as the log size grew, D3 was unable to keep up. To remedy this, we now keep track of where the web client is in the file so we only send the updated data for each request, instead of re-rendering the entire dataset each time.

6. RESULTS

We use D3 to create our visualizations. The following three figures are our visualization results from data collected during lvisor and xv6 booting.

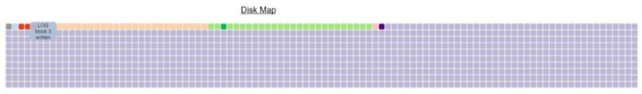


Figure 2: Disk Map after xv6 Boot

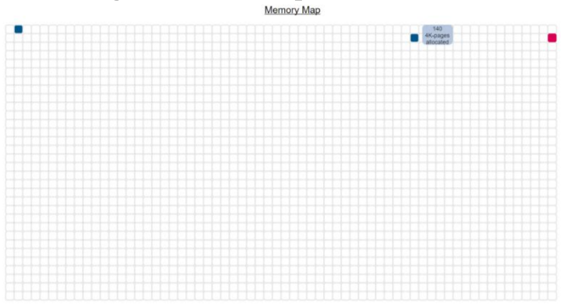


Figure 3: Memory Map after xv6 Boot

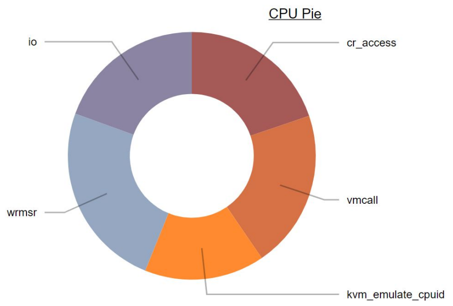


Figure 4: CPU Pie Chart

After booting, xv6 starts shell, and as we run programs in the shell and data is collected during execution of some program is piped to D3, our visualizations dynamically change to reflect the data. For example, we can run the program stressfs, which we used to allocate and deallocate hundreds of 4K-sized pages and also write to multiple disk blocks. As data is piped to D3, blocks in memory that are being written to will blink and the tooltip updates with the number of pages still allocated in a particular page. Blocks in disk will also darken and blink when they have been written to due to a flush in the block cache, and tooltips indicate to the user the region of the disk block. The CPU time chart updates based on the amount of time spent in lvisor due to some VMexit reason.

The following figures are the memory and disk visualizations during execution of stressfs (there was no dramatic change in the CPU time visualization from the previous screenshot):

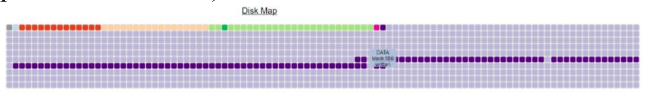


Figure 5: Disk Map after stressfs

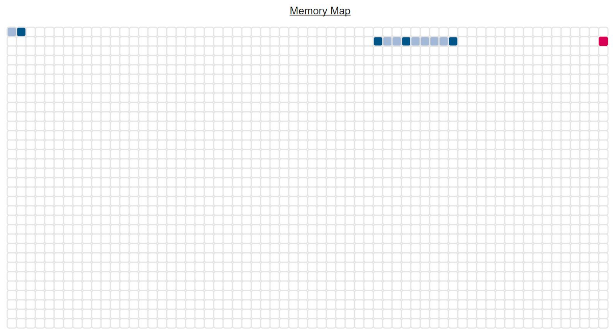


Figure 6: Memory Map after stressfs

The lighter blue boxes in the memory map represent pages in memory that had all of its 4K-page size chunks freed by the guest OS. Since these visualizations dynamically change as a program in the guest OS executes, users quickly see how memory and disk usage and CPU time in relation to the hypervisor changes over time.

7. EXTENSIONS

Due to a copious usage of VMCALL, the performance of the operating system slows down. In addition, the VMCALL approach is not scalable since we have to instrument every guest OS we want to visualize. To improve out design, we could find ways to trigger traps into the lvisor without explicitly inserting VMCALLs inside xv6's code. For example, we can let lvisor change the page table of xv6 to be read-only. Thus whenever xv6 is requesting more memory, it will try to update the page table, trap into the lvisor, and then log information; the amount of VMCALLs will be reduced.

One aim of this project was to change the guest OS, xv6, minimally while still collecting runtime information through the hypervisor. The main techniques behind this would be those used for the console, breaking on key instructions and binary translation, which would make OS-level changes fewer.

Secondly, we should extend this project further to improve the method of export, as file-based techniques rely on concurrent access to a file and unsupported usages of the Python subprocess API. One substitution would be using networking to export from lvisor, adding a TCP stack, e1000 driver, and networking syscalls to connect with the running Flask server to pipe data out of lvisor.

Another extension of the project that was explored and abandoned was supporting multiple operating systems running within lvisor. To do so, we would need to create an array of VM Control Structures (VMCS) to track the state of multiple running operating systems. Furthermore, we would need to implement a scheduler such that each OS gets scheduled. To do so, we investigated usage of Intel's VMX Pre-emption Timer, wherein we could schedule operating systems in a round robin fashion. This would provide additional value to our visualizations as we could make sure contention for resources could be monitored and facilitate development of resource-allocation strategies.

8. CONCLUSION

We want to provide visualization tools for management of virtual machines within a hypervisor. In our capstone project, we collect data on memory, CPU, and disk usage when the hypervisor lvisor runs one guest operating system (xv6). We insert VMCALLs to collect data on those three categories. We log data in a temporary file and periodically parse the log to pipe data categorically to D3 to render visualizations. When different processes are run in the web console, our visualizations dynamically change based on the new data collected for those processes. In this way, we show the interaction of the guest operating system with memory and disk, and we display the time spent in lvisor given a VMexit reason. Our visualizations provide insight on what happens when we run processes within a guest operating system that runs on top of a hypervisor.

A key area of development that was investigated in terms of the interactive console is changing running guest OSs minimally while still visualizing and interacting with them. This is a key concern in any hypervisor - users do not want to run a custom operating system for their hypervisor to get full functionality. We investigated approaches that involve varying levels of changes to the guest OS, although we opted for an approach that did not minimize OS changes.

9. REFERENCES

[1] Bostock, Mike. "Data-Driven Documents." *D3.Js*, d3js.org/

[2] Ellingwood, Justin. "Contents." *How To Use the DigitalOcean Console to Access Your Droplet | DigitalOcean*, DigitalOcean, 27 Feb. 2018, www.digitalocean.com/community/tutorials/how-to-use-the-digitalocean-console-to-access-your-droplet.

[3] Goldschmidt, Simon. "LwIP - A Lightweight TCP/IP Stack - Summary." *LwIP - A Lightweight TCP/IP Stack*, 17 Oct. 2002, savannah.nongnu.org/projects/lwip/.

[4] Ronacher, Armin. "Welcome." *Welcome | Flask (A Python Microframework)*, 16 May 2017, flask.pocoo.org/.