CMSC818s: Project 3
A Brief Survey of Modern Virtualization Research

Greg Benjamin
gregben@cs.umd.edu

Austin Myers
amyers@cs.umd.edu

Department of Computer Science
University of Maryland, College Park

1 Introduction

The concept of virtualization has been around for a long time, reaching back to the 1960s and the introduction of virtual memory in timeshared machines. However, in the last decade or so it's become a very exciting area of research, particularly motivated by a new market for server virtualization. Commerical products like VMWare and open-source platforms like Xen [1] have made it possible for companies to host dozens of instances of their servers on the same hardware. Companies can now spend less on metal, can seamlessly and easily migrate servers from one machine to another, and can even vary their fleet size on the fly in response to fluctuations in server traffic by spinning up new instances. In addition to the commercial successes that have come as a result of server virtualization, there have been tremendous breakthroughs in the research community as well, ranging from new solutions to problems of isolation and protection to novel new algorithms for scheduling and resource allocation.

Despite the flurry of interest in this area, there are still many interesting and difficult problems to be solved. Sharing finite resources fairly and safely amongst many competing processes has never been easy, but the popularity of virtualization in the cloud makes scalability and efficiency bigger concerns than ever before. Furthermore, the commercialization of virtual machines raises questions of how to ensure quality of service in an environment where everyone is in contention for the hardware. In this paper, we will examine some of the recent literature in this area and seek to draw some conclusions about the problem space and the directions of future work.

The rest of the paper is structured as follows: in Section 2 we provide background information on the history of virtualization and the sorts of techniques which are utilized today. Section 3 presents summaries of three recent virtualization papers from the 9^{th} USENIX Symposium on Operating Systems Design and Implementation, as well as a fourth from (TBD!). Section 4 analyses these papers in terms of how they relate to each other and how they approach the larger problem space. Finally, in Section 5 we conclude by discussing our thoughts and opinions on the problem space as a whole, and postulate about how virtualization may evolve in the future.

2 Background

The concept of virtualization reaches back to the 1960s, when developers of large time-sharing systems sought ways to give each user the illusion that they had sole access to a private, virtual computer, despite the fact that many users were in fact sharing computing resources. Once private physical computers became available, popular, and cheap in the 80s, virtualization took a backseat to other areas of research. However, many of the ideas put forth in these early days lived on in work involving virtual memory and operating system protection paradigms.

Server virtualization arrived on the scene about a decade ago, with VMWare appearing in 1999 and Xen in 2003. Since then, research and commercial interest in virtualization has sparked significantly, especially in conjunction with the rise of cloud computing and infrastructure-as-a-service.

In a virtualized context, we have many *virtual machines*(VM) sharing the same physical hardware. In a *fully virtualized* system, each VM has the illusion that it alone has sole access to the hardware, although in a *paravirtualized* VMs may be modified to be aware of the fact that they are virtualized. This sharing of resources is possible because access to the hardware is regulated by a special piece of software, the *virtual machine monitor* or *hypervisor*. In many ways, the hypervisor plays the role of the kernel in a traditional operating system - it contains the privileged software that can directly access the hardware.

The hypervisor (and potentially other administra-

tive software) runs in the *host* domain (in Xen, this is called *dom0*), while other OSes running in VMs are referred to as *guests* (in Xen, all guests run in *domU*). Generally, it is crucial that all guests are isolated from one another, so that the actions of one guest cannot adversely affect the performance of any other. This is a difficult property to provide in a system where all guests are in contention for hardware resources.

In the early days of virtualization, there was little-to-no hardware support, and so the multiplexing of VMs onto the physical hardware was accomplished by *emulating* hardware devices in the software for each VM. This had the unfortunate drawback of being extremely slow and inefficient, because the hardware was emulated by the hypervisor, and so most device accesses by a guest VM resulted in multiple traps into the hypervisor. In the past decade or so, chipsets have become more friendly, and most architectures (notably the x86) now offer hardware-level support for virtualization, along with a set of privileged instructions for making use of it. This has allowed hardware virtualization to become fast enough to be viable, so that virtualized systems are now reasonably competitive with their native counterparts.

3 Paper Summaries

3.1 The Turtles Project: Design and Implementation of Nested Virtualization

The Turtles Project [2] aims to tackle the problem of nested virtualization; rather than being able to run multiple guest OSes on a single host machine, the authors want to take things a step farther and be able to run multiple guest hypervisors. This is actually a much harder problem than its single-level counterpart, because while most hardware now supports virtualization, chips are generally designed to only handle a single level. Furthermore, the strawman solution of using software to emulate the virtualization-enabled hardware for the next level is extremely expensive, because any kernel traps generated at the highest level (from memory paging or device I/O) generate traps in the levels below as well, resulting in cascading crossings of protection boundaries and a serious performance penalty.

Key contributions of this work include the design and implementation of a truly efficient nested VM for the x86 architecture, including novel techniques for *multidimensional paging* and *multi-level device assignment*. The Turtles hypervisor is able to run unmodified guest hypervisors and OSes with a performance penalty of as low as 6-8%. The paper also presents what the authors claim is the first thorough analysis and evaluation of nested hypervisor performance, with an eye to the identification and reduction of performance bottlenecks.

Turtles is designed for the x86 architecture, which subscribes to a single-level hardware support model. This means that there are only two levels of execution privilege. Turtles runs its host hypervisor, which we will call L_0, at the most privileged level of execution. Now, suppose we have a hypervisor running above L_0 at level L_1, and that hypervisor is running a guest OS at level L_2. Everything running at L_1 and L_2 has to run at the same privilege level, and only L_0 is privileged enough to interact with the hardware. Essentially, levels L_1 through L_n only exist logically, and in reality they are all multiplexed to be running directly on top of L_0. All traps generated at L_2 go immediately to L_0, which may handle them itself, or forward them back to L_1 for handling if hardware interaction is not required.

There are two difficult problems here that turtles provides a unique solution for. The first involves page tables for the MMU. In a nested model, virtual addresses for L_2 will need to be translated three times (first to physical addresses at L_2, then to L_1 addresses, and finally to L_0 addresses) before they are readable by the hardware. Modern MMUs provide support for virtualization in the form of two distinct translation tables to make this process fast, but this is not sufficient for more than one level of virtualization. Furthermore, providing these translation tables by emulating them in software is slow, particularly on updates, for the reasons described above. The paper's novel solution here is to use *multidimensional page tables*. Essentially, each level keeps its own shadowed page tables in software, but they are read-only. On a page fault in L_2, L_0 catches the trap and forwards it to L_1. If L_1 also faults, L_0 allows it to update its shadow table with the L_2-to-L_1 translation, but then applies its tranlation for L_1 addresses to update the hardware table with the correct L_2-to-L_0 translation before returning. In this way, the hardware table at L_0 contains all the shadow tables above it compressed into a single table, so that any MMU accesses can look up the correct translations directly from the hardware without having to go through cascading levels of shadow tables first. Turtles also employs "huge pages" and Virtual Processor ID tags on page table entries to ensure that page tables remain small and fast.

The second difficulty deals with device I/0. Traditional approaches to this problem involve either software emulation (which is slow), installing modified drivers in the guest (which is not preferrable),

or assigning devices to guests as they have need of them (which is hard in the nested case). Turtles uses something called *multi-level device assignment*, which is essentially an adaptation of the multidimensional page table idea to the device I/O problem. Modern chipsets include a component called an IOMMU, which resides between the DMA and main memory and performs translations just like the MMU does for the CPU. By shadowing read-only IOMMU translation tables at higher levels, and then compressing them into a single hardware-level table, the DMA can avoid the same cascading translation issue that would affect the MMU, thus making DMA I/O access very efficient. For Memory-Mapped I/O and Network Port I/O, address translation is not an issue, and so here techniques for single-level device assignment are used without any penalty.

The Turtles paper provides an extensive evaluation section, with thorough experimentation comparing performance of applications running on the native OS, a single-level VM, and the Turtles two-level nested VM. The authors measure and discuss performance on microbenchmarks, I/O intensive workloads, and workloads that are designed to fault to the kernel constantly. The paper also analyzes the success of multidimensional paging and multi-level device assignment by testing these techniques against more traditional ways of handling these problems. In general, the Turtles hypervisor far outperforms any other approach to nested virtualization considered, and is able to minimize the overhead of a second level of virtualization to as low as 6-8%, which is rather impressive. No experiments were conducted involving more than two nested levels of virtualization, although analysis seemed to indicate that most of the additional overhead incurred by adding a second level came from cache pollution and the overuse of privileged instructions where it was not necessary.

3.2 mClock: Handling Throughput Variability for Hypervisor IO Scheduling

The mClock paper [3] is not targeted at nested virtualization, but further explores the issue of fair I/O device assignment raised in the Turtles paper. This turns out to be a very difficult problem even in the single-level case, particularly when applied to virtualization in the cloud, where networked devices are often shared not just between VMs running on one host, but between many hosts across the network. In the interest of fairness, a good hypervisor will try to share device throughput equally across its guests, but in the networked case, the total amount of capacity to share may fluctuate due to traffic from other networked clients. Furthermore, many OSes run their own device access algorithms above the hypervisor level to improve performance; for example, physical disk accesses may be reordered by the kernel to take advantage of spatial locality when moving the disk head. These sorts of optimizations are often difficult for the hypervisor to multiplex without loss of efficiency.

The mClock algorithm seeks to fairly schedule the device I/O of multiple guests, while maintaining each guest's limit (maximum throughput requirement), reservation (minimum requirement), and weight (proportional share of capacity relative to other guests). The algorithm uses a "novel, lightweight" tagging scheme to achieve this despite fluctuation in total capacity. Previous attempts to solve this problem were not able to achieve both of these goals simultaneously.

The mClock algorithm works by assigning numerical tags to each queued request. Essentially, tags for VM i with proportional weight w_i will be spaced by $1/w_i$. Requests will be executed in the order of their assigned tags, which ensures that VM i receieves his proportional share. This approach requires a global clock for tag assignment, so that idle clients remain synced with active ones and do not amass "idle credit", thus gaining preferential treatment. The tag for VM i and request j is set as follows:

$$Tag(i,j) = max(curr\ time, Tag(i,j-1) + 1/w_i)$$

After the tagging, the request with the lowest tag will be scheduled.

In actuality, things are a little bit more complicated. mClock maintains 3 types of tags and 3 global clocks, one each for reservations, weights, and limits. Each iteration of the algorithm is deemed to be either a "constraint-based phase", in which some VM has not received capacity above its reservation, or a "weight-based phase", in which the next VM to be scheduled will be chosen by proportional share. This is easily determined after the tagging: if any VM i has request j such that the reservation tag $R(i,j)$ is less than the current time, than VM i is currently receiving less capacity than its reservation, and so the current phase is constraint based. Otherwise, the phase is weight-based, and so the algorithm just picks a request with minimum weight tag $P(i,j)$ whose limit tag $L(i,j)$ is also less than the current time (otherwise, VM i is receiving more capacity than its limit, and so it should not be scheduled).

The paper describes some further tweaks to the algorithm, such as allowing a limited amount of idle credit to be saved up by idle VMs, or breaking "large" requests into a series of smaller requests to account

for request latency in the scheduler. There's also a modifed dmClock algorithm for use with a distributed storage system, which multiplexes VM requests fairly across multiple storate servers. Finally, there's a heuristic for setting reservations for all VMs by default, so that no VM will be starved.

The authors include a lengthy evaluation of the performance of their algorithm. They discuss many experiments involving different workloads and different combinations of weights, limits, and reservations, and even test multiple types of devices being assigned to analyze the effects of device latency and varying throughput. Overall, mClock is extremely effective at meeting its guarantees for weights, limits, and reservations when there is enough capacity to satisfy all requests, and the algorithm degrades gracefully by sharing capacity proportionally when the total capacity is insufficient.

The algorithm is successful at delivering on its guarantees despite contention, which is important for ensuring VM isolation, and also has implications in the cloud, where meeting quality of service guarantees determine financial success.

3.3 Virtualize Everything but Time

The "Virtualize Everything but Time" paper [4] notes that accurate timekeeping has become very important in today's computing systems, with applications to everything from systems measurement and distributed database consistency to high-speed trading and billing for cloud services. However, timekeeping in a VM context is very difficult. Historically, timekeeping is accomplished using software clocks based on hardware oscillators, and keeping the software clocks synchronized to an online reference clock. However, the synchronization algorithms used rely on computing delays between timestamped packets, and the delay computation turns out to be extremely unstable in a virtualized context, due mostly to added latency during packet processing.

This paper does not really introduce a novel idea, but rather applies an existing alternative solution for timekeeping to the virtualized context, and then performs some rigorous experimentation to show that this solution is better-suited.

In the hardware, there are a few oscillators that may be used as sources for timekeeping. Most systems today make use of the *Time Stamp Counter* (TSC), a high-resolution counter based on CPU cycles, and the *High Precision Event Timer* (HPET), which (ironically enough) has a lower resolution than the TSC, and is slower to access as well. The authors mention hardware counters only because the technique currently used for timekeeping in virtualized systems, Xen's Clocksource algorithm, relies on these two counters, using the HPET for low-resolution "ticks", but interpolating between them using the TSC. However, this TSC data must be carefully adjusted before use, because it is subject to drift based on a variety of other variables.

The focus of the paper is more on timekeeping synchronization algorithms. The most popular contender in this arena is the *Network Time Protocol* (NTP). NTP is a feedback algorithm, which works by collecting and timestamping a series of packets from a networked reference clock, computing clock error using the reference timestamp-local timestamp pairings as data points, and updating the system clock accordingly. Xen Clocksource makes use of NTP, although it must be adapted to fit the virtualized context. There are two possibilities for this: in Dependent Clocksource, *dom0* runs an NTP clock, and all guests running in *domU* sync periodically to this NTP clock and use Clocksource to interpolate between syncs. Alternatively, there is an Independent Clocksource paradigm in which each *domU* guest runs their own NTP clock. The paper conducts evaluation of both paradigms, and argues that neither is well-suited to the virtualized context. Dependent Clocksource exhibits cyclical sawtoothed drift because the Clocksource algorithm in *domU* does not have direct access to the TSC, and so cannot adjust the NTP clock data intelligently. Independent Clocksource is extremely inefficient, but even worse, additional latencies incurred by contention for network I/O tend to push the Clocksource feedback algorithm into instability, so that large error can accumulate even across NTP synchronizations.

The authors instead endorse an alternative sync algorithm, *RADClock* (Robust Absolute and Difference Clock). RADClock is a stateless, feed-forward algorithm. Like NTP, RADClock timestamps reference packets, but does so using a "raw" clock based on hardware counters. The raw clock's error is estimated as before, but rather than using this data to update the clock (which would make RADClock a feedback algorithm), the error is instead stored in memory, along with some other clock parameters for the period of the counter and an offset for the timescale being used. When the clock is read, the time is adjusted as

$$C(t) = Raw(t) * (ctr\ period) + (offset) - error(t)$$

RADClock adjusts equally well to either the dependent or equally dependent paradigm, since no interpolation is needed to maintain the clock between syncs, as long as all dependent clocks can look up accurate and up-to-date parameters.

The paper proceeds to rigorously evaluate the performance of Xen Clocksource using NTP against RADClock, both in dependent and independent modes of operation, and in a variety of different system conditions. The conclusion seems to be that RADClock is generally unaffected by the errors and drift that plague clocksource. This is especially clear in a series of experiments conducted involving VM migrations, where "migration shock" pushes the Independent Clocksource clock into a state of instability that takes nearly 20 minutes to correct, while dependent RADClock converges instantly. Furthermore, the RADClock algorithm works equally well regardless of the choice of underlying hardware counter, and is much less affected by high contention for the network and system resources, making it a better solution for the problem of timekeeping in the context of virtualization.

3.4 Paper 4 - TBD

4 Paper Relationships

5 Conclusion

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield. Xen and the Art of Virtualization. In *Proc. of the 19th ACM Symp. on Operating Systems Principles*, 2003.

[2] M. Ben-Yehuda, M. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman and B. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proc. of the 9th USENIX Symp. on Operating Systems Design & Implementation*, 2010.

[3] A. Gulati, A. Merchant, and P. Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *Proc. of the 9th USENIX Symp. on Operating Systems Design & Implementation*, 2010.

[4] T. Broomhead, L. Cremean, J. Ridoux, and D. Veitch. Virtualize Everything but Time. In *Proc. of the 9th USENIX Symp. on Operating Systems Design & Implementation*, 2010.