Virtualization Performance: Zones, KVM, Xen

At Joyent we run a high-performance public cloud based on two different virtualization technologies: Zones and KVM. We have historically run Xen as well, but have phased it out for KVM on SmartOS. My job is to make things *go fast*, which often means using DTrace to analyze the kernel, applications, and those virtualization technologies. In this post I'll summarize their performance in four ways: characteristics, block diagrams, internals, and results.

Attribute	Zones	Xen	KVM
CPU Performance	high	high (with CPU support)	high (with CPU support)
CPU Allocation	flexible (FSS + "bursting")	fixed to VCPU limit	fixed to VCPU limit
I/O Throughput	high (no intrinsic overhead)	low or medium (with paravirt)	low or medium (with paravirt)
I/O Latency	low (no intrinsic overhead)	some (I/O proxy overhead)	some (I/O proxy overhead)
Memory Access Overhead	none	some (EPT/NPT or shadow page tables)	some (EPT/NPT or shadow page tables)
Memory Loss	none	some (extra kernels; page tables)	some (extra kernels; page tables)
Memory Allocation	flexible (unused guest memory used for file system cache)	fixed (and possible double-caching)	fixed (and possible double-caching)
Resource Controls	many (depends on OS)	some (depends on hypervisor)	most (OS + hypervisor)
Observability: from the host	highest (see everything)	low (resource usage, hypervisor statistics)	medium (resource usage, hypervisor statistics, OS inspection of hypervisor)
Observability: from the guest	medium (see everything permitted, incl. some physical resource stats)	low (guest only)	low (guest only)
Hypervisor Complexity	low (OS partitions)	high (complex hypervisor)	medium
Different OS Guests	usually no (sometimes possible with syscall translation)	yes	yes

There are variations with how these can be configured, and details in this table may vary. At the very least, this can serve as a checklist of characteristics to confirm, which may also be helpful if you are considering other technologies (eg, VMWare). Wikipedia also has a table of general characteristics.

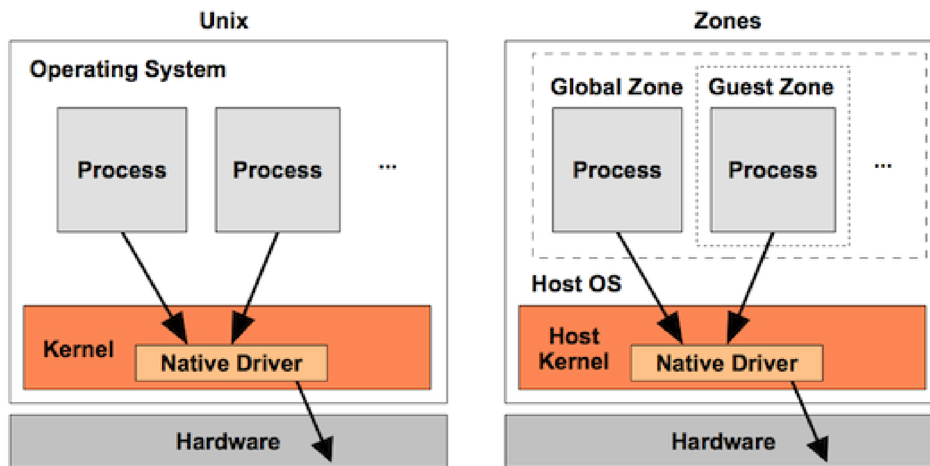
The three in this table represent different types: OS Virtualization (Zones), and Hardware Virtualization of both Type 1 (Xen) and Type 2 (KVM) varieties.

The delivered performance of these is critical. In general, we use fast server hardware, 10 GbE networks, ZFS for all file systems, DTrace for performance analysis, and Zones wherever possible. We also performed our own port of KVM to illumos, and run KVM instances *inside* Zones, providing additional resource controls than can be applied, and improved security ("double-hulled virtualization").

There are many characteristics I'd like to discuss in more detail. In this post, I'll look at the I/O path (network, disk) and its overhead.

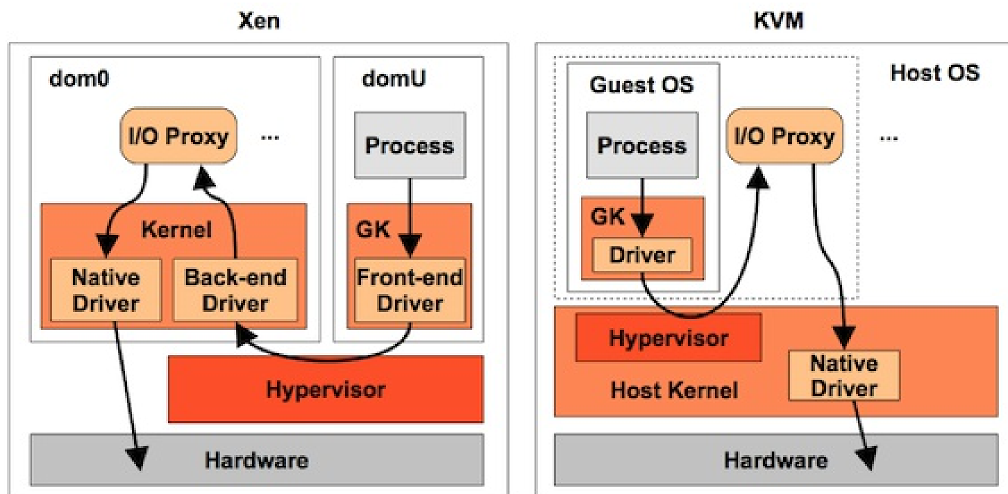
I/O Path

How does I/O differ between traditional Unix and Zones?



Performance is exactly the same – there is no overhead. Zones partition the OS in the same way that chroot isolates a process in the file system. There isn't necessarily an extra layer in the software stack to make this work.

Now for Xen and KVM (simplified!):



GK is Guest Kernel, and domU on Xen runs the guest OS. Some of these arrows are indicating the *control-path*, where components inform each other, either synchronously or asynchronously, that more data is ready to transfer. The *data-path* may be implemented in some cases by shared memory and ring buffers. There are also different ways this can be configured. For example, Xen can use Isolated Driver Domains (IDD), or stub-domains, to run the I/O proxies in isolation.

With **Xen**, the hypervisor performs CPU scheduling for the domains, and then each domain has its own OS kernel for thread scheduling. The hypervisor supports different CPU scheduling classes, including Borrowed Virtual Time (BVT), Simple Earliest Deadline First (SEDF), and Credit-Based. The domains use the OS kernel scheduler, and whatever regular scheduler classes and policies they provide.

The extra overhead of multiple schedulers costs performance. Having multiple schedulers can also create complex issues with how they interact, adding CPU latency in the wrong situations. Debugging this can be very difficult, especially since the Xen hypervisor is running out of reach of the usual OS performance tools (try

xentrace instead).

Sending I/O via the I/O proxy processes (which are usually qemu) involves context-switching and more overhead. There has been lots of work to minimize this, including shared memory transports, buffering, I/O coalescing, and paravirtualization drivers.

With **KVM**, the hypervisor is a kernel module (kvm) which is scheduled by the OS scheduler. It can be tuned using the usual OS kernel scheduler classes, policies and priorities. The I/O path takes fewer steps than Xen. (The original Qumranet KVM paper described it as five steps vs ten, although this description isn't including paravirtualization.)

With **Zones**, there's no comparison. The I/O path – which for high-speed networking is very sensitive – has none of these extra steps. While this has been well known in the Solaris community for years (Zones being a Solaris technology), and also the FreeBSD community (as Zones are based on FreeBSD jails), the Linux community is still learning about them and developing their own version: **Linux Containers**. Glauber Costa described them in his talk "The failure of Operating Systems, and how we can fix it" for Linuxcon 2012, and listed various use cases where KVM was currently used. Many of the use cases could be served by Containers, and didn't actually need KVM.

Sometimes you (and our customers) really do need Hardware Virtualization, as their applications depend on a particular version of the Linux kernel, or Windows. We provide this with KVM (we've phased out Xen).

Internals

Some deeper insights into how these work (often using DTrace).

Network I/O, Zones

The following two stack traces show how a network packet is transmitted from the global zone (the host, which is the same as a bare-metal install) and from a zone (the guest):

```
Global Zone:                            Zone:
mac`mac_tx+0xda                         mac`mac_tx+0xda
dld`str_mdata_fastpath_put+0x53         dld`str_mdata_fastpath_put+0x53
ip`ip_xmit+0x82d                        ip`ip_xmit+0x82d
ip`ire_send_wire_v4+0x3e9               ip`ire_send_wire_v4+0x3e9
ip`conn_ip_output+0x190                 ip`conn_ip_output+0x190
ip`tcp_send_data+0x59                   ip`tcp_send_data+0x59
ip`tcp_output+0x58c                     ip`tcp_output+0x58c
ip`squeue_enter+0x426                   ip`squeue_enter+0x426
ip`tcp_sendmsg+0x14f                    ip`tcp_sendmsg+0x14f
sockfs`so_sendmsg+0x26b                 sockfs`so_sendmsg+0x26b
sockfs`socket_sendmsg+0x48              sockfs`socket_sendmsg+0x48
sockfs`socket_vop_write+0x6c            sockfs`socket_vop_write+0x6c
genunix`fop_write+0x8b                  genunix`fop_write+0x8b
genunix`write+0x250                     genunix`write+0x250
genunix`write32+0x1e                    genunix`write32+0x1e
unix`_sys_sysenter_post_swapgs+0x14     unix`_sys_sysenter_post_swapgs+0x14
```

I spent (way) too much time double-checking that I didn't switch these two stacks by accident, since they are *identical*. The stack on the right shows the same code path taken.

You could configure Zones in a way that it does have overhead, just like on a normal system. For example, enabling a firewall for network I/O, or mounting file systems via lofs instead of directly. These are optional, and may be worth the extra performance overhead for certain use cases.

Network I/O, KVM

The full code path for performing network I/O is complex.

The first part is the guest process writing to its driver. In this case, I'm demonstrating a **Linux** Fedora guest

with DTrace-for-Linux, and tracing the paravirt driver:

```
guest# dtrace -n 'fbt:virtio_net:start_xmit:entry { @[stack(100)] = count(); }'
dtrace: description 'fbt:virtio_net:start_xmit:entry ' matched 1 probe
^C
[...]
              kernel`start_xmit+0x1
              kernel`dev_hard_start_xmit+0x322
              kernel`sch_direct_xmit+0xef
              kernel`dev_queue_xmit+0x184
              kernel`eth_header+0x3a
              kernel`neigh_resolve_output+0x11e
              kernel`nf_hook_slow+0x75
              kernel`ip_finish_output
              kernel`ip_finish_output+0x17e
              kernel`ip_output+0x98
              kernel`__ip_local_out+0xa4
              kernel`ip_local_out+0x29
              kernel`ip_queue_xmit+0x14f
              kernel`tcp_transmit_skb+0x3e4
              kernel`__kmalloc_node_track_caller+0x185
              kernel`sk_stream_alloc_skb+0x41
              kernel`tcp_write_xmit+0xf7
              kernel`__alloc_skb+0x8c
              kernel`__tcp_push_pending_frames+0x26
              kernel`tcp_sendmsg+0x895
              kernel`inet_sendmsg+0x64
              kernel`sock_aio_write+0x13a
              kernel`do_sync_write+0xd2
              kernel`security_file_permission+0x2c
              kernel`rw_verify_area+0x61
              kernel`vfs_write+0x16d
              kernel`sys_write+0x4a
              kernel`sys_rt_sigprocmask+0x84
              kernel`system_call_fastpath+0x16
             2015
```

That's the Linux 3.2.6 network transmit path.

Control is passed by KVM to the qemu I/O proxy, which then transmits it on the host OS via the usual means (native driver). Here is the **SmartOS** stack in this case:

```
host# dtrace -n 'fbt::igb_tx:entry { @[stack()] = count(); }'
dtrace: description 'fbt::igb_tx:entry ' matched 1 probe
^C
[...]
              igb`igb_tx_ring_send+0x33
              mac`mac_hwring_tx+0x1d
              mac`mac_tx_send+0x5dc
              mac`mac_tx_single_ring_mode+0x6e
              mac`mac_tx+0xda
              dld`str_mdata_fastpath_put+0x53
              ip`ip_xmit+0x82d
              ip`ire_send_wire_v4+0x3e9
              ip`conn_ip_output+0x190
              ip`tcp_send_data+0x59
              ip`tcp_output+0x58c
              ip`squeue_enter+0x426
              ip`tcp_sendmsg+0x14f
              sockfs`so_sendmsg+0x26b
              sockfs`socket_sendmsg+0x48
              sockfs`socket_vop_write+0x6c
              genunix`fop_write+0x8b
              genunix`write+0x250
              genunix`write32+0x1e
              unix`_sys_sysenter_post_swapgs+0x149
             1195
```

Both of these stacks are pretty complex to begin with. Then there is the stuff in-between the **Linux kernel** and the **illumos kernel**, which gets even more complicated and involved. Basically, the paravirt code paths allow the two kernel stacks to make intimate love.

When Robert Mustacchi of Joyent last investigated these code paths in detail, he drew up some wonderful ASCII diagrams like the following:

```
/*
 *                GUEST                      #       QEMU
 *
##################################################################
 *                                     #
 *   +----------+                            #
 *   |  start_  | (1)                        #
 *   |  xmit()  |                            #
 *   +----------+                            #
 *       ||                                  #
 *       ||       +-----------+              #
 *       ||------>|free_old_  | (2)          #
 *       ||------>|xmit_skbs()|              #
 *       ||       +-----------+              #
 *       \/                      (3)         #
 *   +---------+       +-------------+     + - #--- PIO write to VNIC
 *   |  xmit_  |------->|virtqueue_add|     |   #    PCI config space (6)
 *   |  skb()  |------->|_buf_gfp()   |     |   #
 *   +---------+       +-------------+     |   #
 *       ||                                |   # +- VM exit
 *       ||        +- iff interrupts       |   # |  KVM driver exit (7)
 *       \/        |  unmasked (4)         |   # |
 *   +---------+   |     +-----------+(5)  |   # |  +---------+
 *   |virtqueue|----*---->|vp_notify()|-----*---#-*->| handle  | (8)
 *   |_kick()  |----*---->|           |-----*---#-*->|PIO write|
 *   +---------+         +-----------+        #    +---------+
 *       ||                                  #       ||
 *       ||   (13)                           #       ||
 *       **-----+ iff avail ring             #       \/     (9)
 *       ||      capacity < 20               # +-----------------+
 *       ||      else return                 # |virtio_net_handle|
 *       ||                                  # |tx_timer()       |
 *       \/   (14)                           # +-----------------+
```

```
*   +----------+                        #  ||
*   |netif_stop|                        #  ||          (10)
*   |_queue()  |                        #  ||   +---------+
*   +----------+                        #  ||-->|qemu_mod_|
*       ||                              #  ||-->|timer()  |
*       ||    (15)              (16)    #  ||   +---------+
*   +----------------+    +----------+      #  ||
*   |virtqueue_enable|---->|unmask    |      #  ||          (11)
*   |_cb_delayed()  |---->|interrupts|      #  ||  +------------+
*   +----------------+    +----------+      #  |+->|virtio_     |
*       ||                ||            #  +-->|queue_set_  |
*       || (18)           ||      (17)    #     |notification|
*       ||  +-return   +-------------------+  #     +------------+
*       ||  | iff ---->|check if the number|  #      |
*       **--+ is false |of unprocessed used|  #      |  disable host
*       ||          |ring entries is >  |  #      +- interrupts
*       ||          |3/4s of the avail  |  #         (12)
*       \/   (19)     |ring index - the   |  #
*  +-----------+      |last freed used    |  #
*  |free_old_  |      |ring index         |  #
*  |xmit_skbs()|      +-------------------+  #
*  +-----------+                        #
*       ||                              #
*       ||    (20)                        #
*       **-----+ iff avail ring              #
*       ||      capacity is                #
*       ||      now > 20                   #
*       \/                              #
*  +-----------+                        #
*  |netif_start| (21)                     #
*  |_queue()   |                        #
*  +-----------+                        #
*       ||                              #
*       ||                              #
*       \/  (22)            (23)          #
*  +------------+     +----------+          #
*  |virtqueue_  |----->|mask      |          #
*  |disable_cb()|----->|interrupts|          #
*  +------------+     +----------+          #
*                                       #
*                                       #
*/
```

Figure II: Guest / Host Packet TX Part 1

I included this diagram just to give you a sense of what happens. And that's only part *1*.

In brief, this uses ring buffers in shared memory to transfer the data, and a notification mechanism to inform when data is ready to transfer. When everything is working as intended, performance can be quite reasonable. It isn't bare-metal fast (or Zones fast), but it isn't terrible either. I've included some numbers later in this post.

The CPU overhead and reduced network performance is one thing. Another is the complexity this introduces, which hampers analysis and performance investigations. With Zones, there is one kernel TCP/IP stack to study and tune. Given its complexity, one is more than enough! With KVM, there are two different kernel TCP/IP stacks, plus KVM and paravirt. Investigating performance can take ten times longer, or so long that it becomes prohibitive. This is why I included "Observability" as a key characteristic in my comparison table. If it's harder to see, it's harder to tune.

Network I/O, Xen

The guest transmit and I/O proxy transmit stacks are the same. The in-between bit gets more complex. The hypervisor can't be inspected using OS observability and debugging tools, since it's running on bare-metal directly. There is xentrace, which looks pretty useful, as it instruments many event types in the Xen scheduler using static probes. (Even if it isn't real-time and programmatic like DTrace, and, requires me to learn Yet Another Tracer.)

/proc, Zones

While the I/O path may have zero extra overhead by default, there are some overheads with OS Virtualization, usually for administration or observability, and not in the CPU or I/O "hot path".

For example, a Zone cannot see other guests on the same system via /proc, as read by `prstat(1M)`, `top(1)`, etc. This is implemented in usr/src/uts/common/fs/proc/prvnops.c:

```
static int
pr_readdir_procdir(prnode_t *pnp, uio_t *uiop, int *eofp)
{
[...]
        /*
         * Loop until user's request is satisfied or until all processes
         * have been examined.
         */
        while ((error = gfs_readdir_pred(&gstate, uiop, &n)) == 0) {
                uint_t pid;
                int pslot;
                proc_t *p;

                /*
                 * Find next entry.  Skip processes not visible where
                 * this /proc was mounted.
                 */
                mutex_enter(&pidlock);
                while (n < v.v_proc &&
                    ((p = pid_entry(n)) == NULL || p->p_stat == SIDL ||
                    (zoneid != GLOBAL_ZONEID && p->p_zone->zone_id != zoneid) ||
                    secpolicy_basic_procinfo(CRED(), p, curproc) != 0))
                        n++;
[...]
```

The full list of processes are scanned, and just the local Zone's processes are returned. This might sound a bit inefficient – couldn't a linked list be added to `proc_t` so that Zone processes could be walked directly? Sure, but let's be data driven.

Here's the time to read /proc from a Zone by the `prstat(1M)` command, measuring using DTrace:

```
# dtrace -n 'fbt::pr_readdir_procdir:entry /execname == "prstat"/ {
    self->ts = timestamp; } fbt::pr_readdir_procdir:return /self->ts/ {
    @["ns"] = avg(timestamp - self->ts); self->ts = 0; }'
dtrace: description 'fbt::pr_readdir_procdir:entry ' matched 2 probes
^C
  ns                                                        544584
```

On average, that's 544 us (microseconds).

Now with an extra 1000 processes in another Zone (which represents a typical dozen extra guests):

```
# dtrace -n 'fbt::pr_readdir_procdir:entry /execname == "prstat"/ {
   self->ts = timestamp; } fbt::pr_readdir_procdir:return /self->ts/ {
   @["ns"] = avg(timestamp - self->ts); self->ts = 0; }'
dtrace: description 'fbt::pr_readdir_procdir:entry ' matched 2 probes
^C
  ns                                                        594254
```

That added 50 us. For a /proc read – which shouldn't be hot path. If it is, and 50 us matters, we can look at it then.

(While I was here, I also checked `pidlock`, which is, ahem, *global*. It is not currently a problem. This was also

checked using DTrace.)

Network Throughput Results

I try not to share performance testing results without triple checking numbers, and I don't have time for that right now (this was just supposed to be a quick blog post). I can share some previous numbers from a few months ago, when I did have the time to test carefully and perform Active Benchmarking.

This was a series of network throughput and IOPS tests using iperf, to test differences with default installations of 1 Gbyte SmartOS Zones and CentOS KVM instances (Xen wasn't tested). The client and server were in the same datacenter, but not on the same physical host, so that the full network stack was used.

I should make it clear that these results are not a "max config" for our cloud. It's a **minimum config** (1 Gbyte instances). If this were a marketing activity, I'd probably be compelled to test the max config. Which, for our SmartOS kernel, will be a lot of work, as it can drive multiple 10 GbE ports at line rate, which requires a lot of load-generating clients to perform.

For these results, YMMV based on workload, platform kernel type, and tuning. If you are to use them, think carefully about how they would apply, and to what degree. If you workload is CPU- or File System-bound, then you are probably better off testing their performance than using these network results.

A typical invocation on the **server**:

```
iperf -s -l 128k
```

And on the **client**:

```
iperf -c server -l 128k -P 4 -i 1 -t 30
```

The thread count (-P) was varied to investigate limits. The final result – the average over 30 seconds – was used.

Throughput

Searching for the highest Gbits/sec:

source	dest	threads	result	suspected limiter
SmartOS 1 GB	SmartOS 1 GB	1	2.75 Gbits/sec	client iperf @80% CPU, and network latency
SmartOS 1 GB	SmartOS 1 GB	2	3.32 Gbits/sec	dest iperf up to 19% LAT, and network latency
SmartOS 1 GB	SmartOS 1 GB	4	**4.54 Gbits/sec**	client iperf over 10% LAT, hitting CPU caps
SmartOS 1 GB	SmartOS 1 GB	8	1.96 Gbits/sec	client iperf LAT, hitting CPU caps
KVM CentOS 1 GB	KVM CentOS 1 GB	1	**400 Mbits/sec**	network/KVM latency (dest 60% of the 1 VCPU)
KVM CentOS 1 GB	KVM CentOS 1 GB	2	394 Mbits/sec	network/KVM latency (dest 60% of the 1 VCPU)
KVM CentOS 1 GB	KVM CentOS 1 GB	4	388 Mbits/sec	network/KVM latency (dest 60% of the 1 VCPU)
KVM CentOS 1 GB	KVM CentOS 1 GB	8	389 Mbits/sec	network/KVM latency (dest 70% of the 1 VCPU)

The peak **Zones** performance was 4.54 Gbits/sec with 4 threads. More threads hit the CPU caps for the 1 Gbyte (small) instance, with the CPU scheduler latency causing TCP breakdown. Larger SmartOS instances have higher CPU caps, and should be able to take performance further.

For the **KVM** test, these were default CentOS instances. I know that with a more modern Linux kernel with network stack tuning, we can improve throughput much further. The most I've reached is around 900 Mbits/sec for 1 VCPU KVM Linux (this was after we tuned KVM up from 110 Mbits/sec using a lot of DTrace analysis). Even at 900 Mbits/sec, it's still 5x slower than Zones.

Note the "suspected limiter" column. This is essential to confirm what was actually tested, and comes from Active Benchmarking. It means I did performance analysis for *every single result* (including those not listed here to save room). In case you are wondering, it took a full day to perform all tests and analyze each result (again, using DTrace).

IOPS

Searching for the highest packets/sec:

source	dest	threads	result	suspected limiter
SmartOS 1 GB	SmartOS 1 GB	1	14000 packets/sec	client/dest thread count (each thread about 18% CPU total)
SmartOS 1 GB	SmartOS 1 GB	2	23000 packets/sec	client/dest thread count
SmartOS 1 GB	SmartOS 1 GB	4	36000 packets/sec	client/dest thread count
SmartOS 1 GB	SmartOS 1 GB	8	60000 packets/sec	client/dest thread count
SmartOS 1 GB	SmartOS 1 GB	16	**78000 packets/sec**	both client & dest CPU cap
KVM Centos 1 GB	KVM Centos 1 GB	1	1180 packets/sec	network/KVM latency, thread count (client thread about 10% CPU)
KVM Centos 1 GB	KVM Centos 1 GB	2	2300 packets/sec	network/KVM latency, thread count
KVM Centos 1 GB	KVM Centos 1 GB	4	4400 packets/sec	network/KVM latency, thread count
KVM Centos 1 GB	KVM Centos 1 GB	8	7900 packets/sec	network/KVM latency, thread count (threads now using about 30% CPU each; plenty idle)
KVM Centos 1 GB	KVM Centos 1 GB	16	13500 packets/sec	network/KVM latency, thread count (~50% idle on both)
KVM Centos 1 GB	KVM Centos 1 GB	32	**18000 packets/sec**	CPU (dest >90% of the 1 VCPU)

In this case, Zones is 4x the packet rate of KVM. As before, the limiting factor becomes the cloud CPU limits, and I was only testing *small* 1 Gbyte servers. Bigger servers get higher CPU quotas, and all of these numbers should scale higher.

Conclusion

In this post, I summarized performance characteristics of three virtualization technologies – Zones, Xen, and KVM – and then investigated the I/O path in more detail. Zones add no overhead, whereas Xen and KVM do, which could limit network throughput to a quarter of what it could be.

By default we encourage customers to deploy on Zones, for reasons of performance, observability, and simplicity (debuggability). This may mean compiling their applications for <="" a="">SmartOS (our illumos-based OS which hosts the Zones) if they aren't already in the repo. In cases where they absolutely must have Linux or Windows, and the applications can't run elsewhere, then it's Hardware Virtualization (KVM).

There are more performance characteristics to consider that I didn't explore here, except briefly in the summary table, including how CPU allocation and VCPUs work, how memory allocation works and file system caches, and more. These could be topics for follow up posts.

This post wasn't supposed to be so much about DTrace, but it's the essential tool in so much of our high-performance work that it would be hard not to mention. We use it to improve overall performance for Zones and KVM, to track down latency outliers, explain benchmark results, study the effects of multi-tenancy, and to improve the performance of applications and the OS.

Posted on January 11, 2013 at 3:58 pm by Brendan Gregg · Permalink In: Cloud · Tagged with: cloud, dtrace, kvm, performance, xen, zones