Chapter 8

(NEW) Virtual Node Scaling

One of the primary requirements of the NCR is that it support 10,000 or more nodes within a single test BAA §3.3.3.2 •3. Depending on the requirements of the experiment in question, some of these nodes are likely to be virtualized to increase scalability of the range. There are several different approaches to virtualization, all providing different levels of emulation fidelity, scalability, and functionality—and with different resource requirements.

While there is no requirement that all of the virtualized nodes in a large scale test have the same instruction set architecture, tests involving large numbers of "virtual x86" boxes, running on top of x86 physical hardware, is expected to be common and is thus what is addressed in this chapter.

In this report, we consider two broad categories: *OS-level virtualization*, in which some or all of the OS namespaces, data structures, or user-visible structures are virtualized so that each virtual entity (perhaps a "container" with its own filesystem, processes, and devices) sees only those structures bound to itself; and *paravirtualization*, in which an entire machine interface (e.g., BIOS, devices, access to privileged instructions) is presented to a guest operating system from the host virtual machine monitor.[1] OS-level virtualization is typically very lightweight, since it does not have to present high-fidelity emulation of virtual resources (i.e., CPU, RAM, hard disk, network interface cards) to the guest operating system, as is the case for paravirtualization. Consequently, OS-level virtualization may be especially useful for hosting small applications within a test whose main requirements are an isolated filesystem and a distinct network interface with a unique address—such applications may not require specific kernel-level versions, modules, or configuration changes. However, if the test requires virtual machines running a specific operating system, such as Windows—or a high-fidelity machine replication in general—paravirtualization is a necessity despite its higher costs to the host, and resulting lower scalability.

Emulab allows users to include *virtual nodes* (also called *vnodes*) in experiments, and provides several different virtualization technology platforms that support this more general abstraction. (Refer to Section 11.3.1 for a discussion of Emulab's vnode architecture, and Section 11.3.2

[1]A third type of virtualization, *full virtualization*, which includes emulation of the machine instruction set, is not discussed.

for implementation details.)

Although the ratio of virtual nodes to physical nodes supported currently by Emulab may not need to increase dramatically to achieve the goals of the NCR, the resource requirements of virtual nodes will increase for some tests. This will require a new and possibly multi-faceted approach to virtualization. For many tests run on the range, lightweight OS-level virtualization as provided by FreeBSD jails [31] or Linux-based OpenVZ containers [43] may suffice. Emulab currently supports both these implementations under its virtual node abstraction, although the FreeBSD support is aging, and the OpenVZ support is still in beta form. However, the NCR's emphasis on fidelity, potentially involving specific operating systems and versions, requires a more completely virtualized environment, such as that provided by Xen [4] or VMware [58]. Emulab developers have recently added new support for Xen, but it has not been released for production use.

In previous chapters, we have already discussed enhancing core Emulab services, including resource configuration (see Chapter 5) and runtime experiment control (see Chapter 7), to scale to support greater numbers of nodes. In this chapter, however, we focus on performance enhancements that virtual node technologies might or will require. We report network-oriented performance characterization results for two virtualization mechanisms currently supported by Emulab, discuss some potential improvements, and more generally discuss possible solutions to expected future problems that will arise within the NCR context.

8.1 Current Emulab Virtualization Technology Scalability

One of the primary goals of Emulab is to provide high-fidelity link emulation, an important goal that must continue to be satisfied in the presence of multiplexed virtual nodes and links. In this section, we provide results from experiments that stress test the network stack in the presence of virtual links that are fully contained on a single physical host. These experiments help establish how many virtual nodes and/or links Emulab can safely pack onto a single physical host node, and illuminate potential bottlenecks in the software portions of the network stack.

8.1.1 Experiment Setup

Our premises are that virtual nodes without links are useless for networking experiments, and that virtual links without guaranteed characteristics (i.e., bandwidth, delay, loss) are very likely useless in the presence of multiple competing such links. Our experiments pack an even number of vnodes onto a single physical host node; each pair of vnodes (one a client, one a server) has a single shaped virtual link connecting them. For instance, in our experiments with 35 pairs, we created 70 virtual nodes and 35 virtual links. In FreeBSD, each virtual node is a jail; in Linux, an OpenVZ container. Each link endpoint has either a TCP sender or receiver which prints bandwidth statistics that we capture. For FreeBSD, we use `ttcp` to provide a single client-server TCP stream per link; on Linux, we use `iperf`. In each test, we shape all virtual links (using dummynet on FreeBSD and tc on Linux) between virtual nodes with the same bandwidth. Consequently, each test is parameterized by 1) the virtual link bandwidth, and 2) the number of virtual links. In addition to measuring bandwidth, we measure various CPU and memory usage indicators on both FreeBSD and Linux.

(Obviously, this leaves two primary scenarios untested. First, we have not examined the case where multiple virtual links flow between virtual nodes hosted on different physical nodes; i.e., multiplexed virtual links. In this case, traffic shaping happens for each multiplexed link endpoint (as it does when a virtual link is hosted entirely within one physical host), *and* there is additional overhead from the real device interaction and any encapsulation necessary to multiplex packets to isolate them at Layer 2. Second, we do not test the LAN case, in which more than two virtual link endpoints must be part of a bridge, so our tests do not consider any potential overhead in the bridging code which connects virtual link endpoints. However, we do not expect this overhead to be significant, since a well-behaved bridging implementation should act no differently if it includes two ports or n ports, for $n > 2$—unless the two port case is optimized.)

8.1.2 Platform Details

All experiments were conducted on Emulab's `pc3000` nodes, which have these specifications: single 3.0 GHz 64-bit Xeon (forced to run in 32-bit mode) CPU with 1 MB L2 cache and 800 MHz FSB (quad-pumped 200 MHz, meaning 4 accesses per cycle); 2 GB RAM in 4 512 MB modules of type DDR2 400 MHz; and 2 146 GB SCSI hard drives. This platform has a theoretical maximum memory bandwidth of 6.4 GB/s, but since we are running in 32-bit mode, the effective bandwidth could vary between 3.2 and 6.4 GB/s.

FreeBSD jail experiments were conducted on our modified version of FreeBSD 4.10. Linux OpenVZ experiments were conducted using a 2.6.18 kernel patched with the current stable OpenVZ patch and our shaping patches (which provide loss rate and delay shaping), as well as other miscellaneous patches.

8.1.3 FreeBSD Jail Results

(The reader may be interested to know that these FreeBSD results are over five years old, but we have included them to provide a baseline for comparison with the new and unoptimized OpenVZ results in Section 8.1.4. At this time, we have no plans to port our FreeBSD 4.x patches forward to more modern FreeBSDs for a variety of reasons.)

Figure 8.1 and Figure 8.2 show the bandwidth achievable for increasing numbers of communicating vnode pairs. These demonstrate that the FreeBSD implementation can only sustain 8 virtual links at full 100 Mbps fidelity. For some topologies, emulated links with lower bandwidths might be useful. However, even though the FreeBSD results only consider up to 10 node pairs (or 10 virtual links), one can observe from the downward trend of the "unlimited" bandwidth (i.e., unshaped) curve at higher numbers of pairs, that the implementation will likely not scale to even 15 virtual node pairs with links shaped at 50 Mbps.

From Figure 8.4, a graph of memory utilization, it is fairly clear that the limiting factor is not the amount of RAM in the physical host. A much more obvious bottleneck is the CPU: Figure 8.3, a graph of CPU utilization shows that the CPU is nearly saturated for 100 Mbps links at between 8 and 10 virtual node pairs. Moreover, the CPU utilization curves are exponential in nature, and will not scale ideally in the face of additional contention at higher numbers of virtual node pairs.

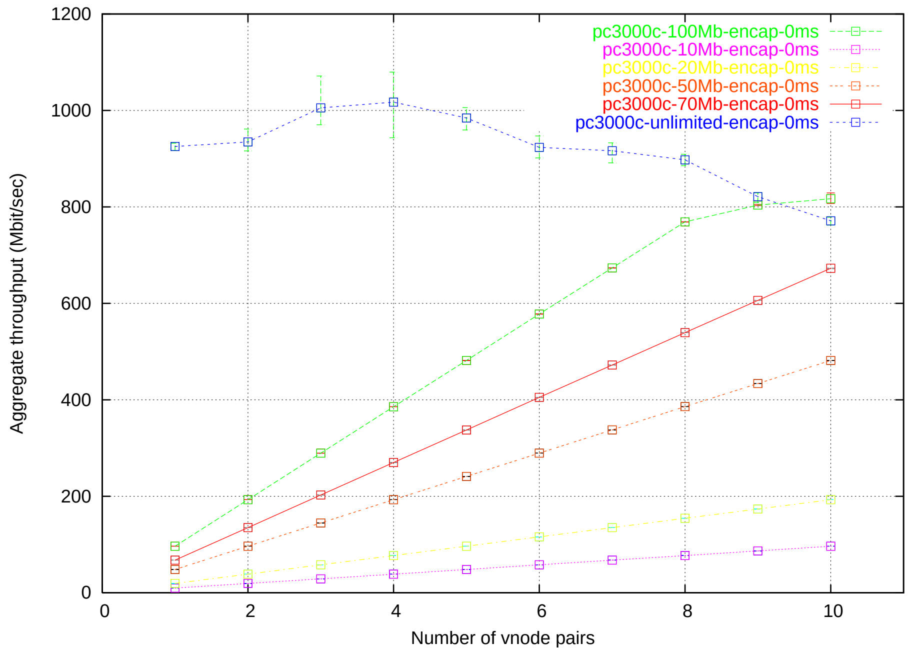

Figure 8.1. (NEW) FreeBSD: Aggregate bandwidth across all virtual links as a function of bandwidth shaping values and number of client-server pairs

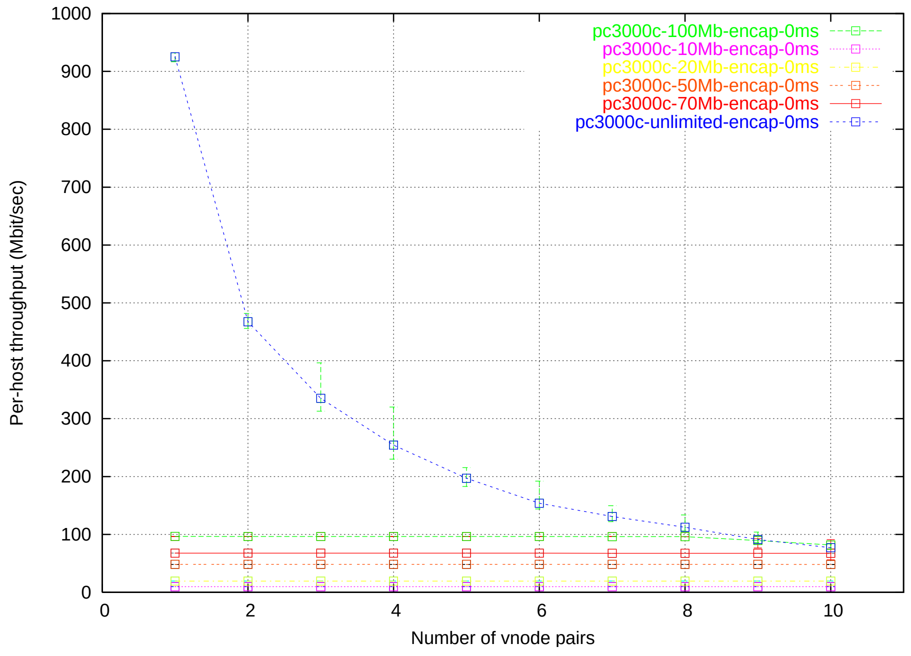

Figure 8.2. (NEW) FreeBSD: Mean bandwidth of each virtual link as a function of bandwidth shaping values and number of client-server pairs

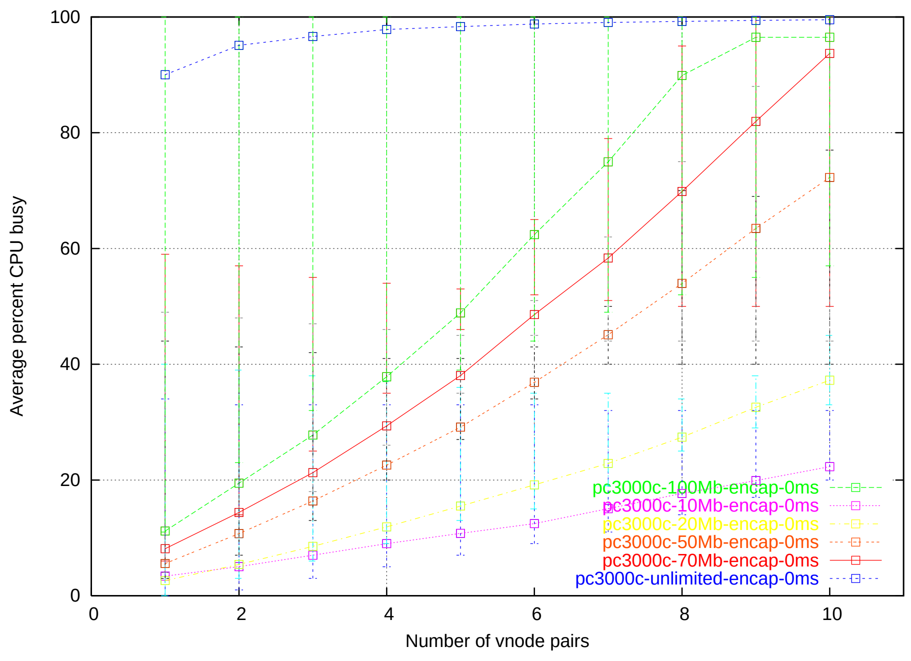

Figure 8.3. (NEW) FreeBSD: Mean percentage of CPU used by the FreeBSD kernel as a function of bandwidth shaping values and number of client-server pairs

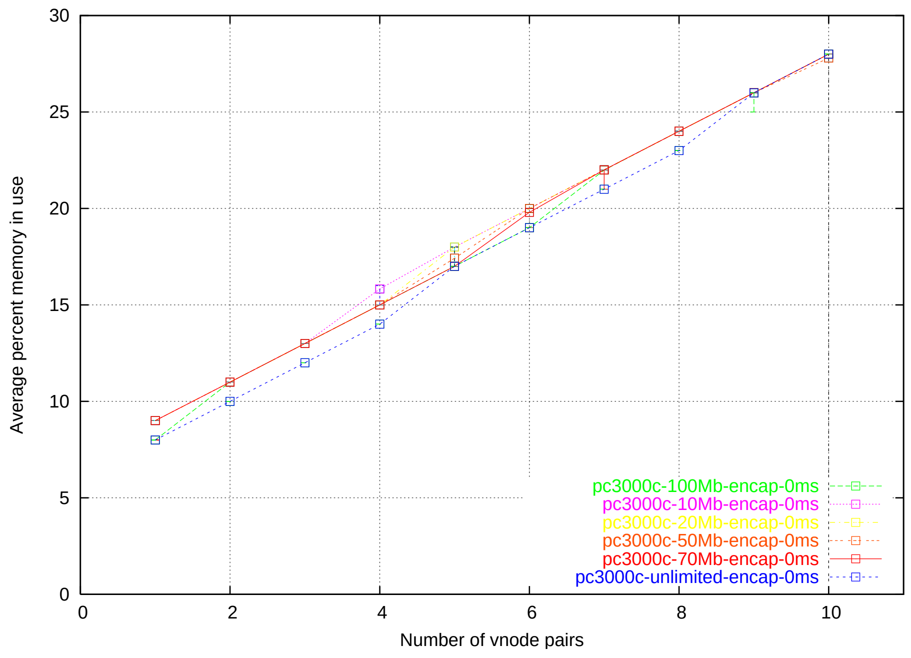

Figure 8.4. (NEW) FreeBSD: Mean physical memory usage as a function of bandwidth shaping values and number of client-server pairs

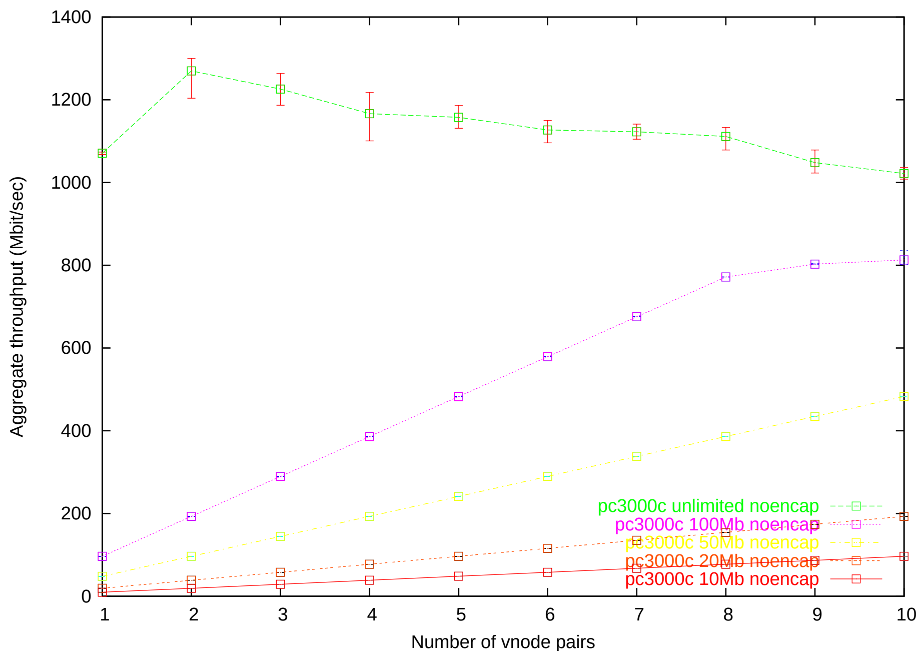

Figure 8.5. (NEW) FreeBSD: Aggregate bandwidth across all virtual links as a function of bandwidth shaping values and number of client-server pairs—with all virtual links operating in non-encapsulating mode

Finally, the Emulab FreeBSD implementation of *veths* (virtual ethernet devices) provides different methods of virtual link multiplexing; this is only really useful when packets must flow over virtual links multiplexed over (i.e., multiple virtual links sharing) a single physical link. Emulab provides several mechanisms to accomplish this: VLAN tags, a custom encapsulation header similar to VLAN tagging based on MAC addresses, or no encapsulation (in which case the switch must associate the fake MAC addresses of the virtual link endpoints with the physical switch ports over which they are multiplexed—which means that virtual links sharing the same physical link are not fully isolated from one another for broadcast Layer 2 traffic). The first four graphs we include show results with encapsulation enabled, but this adds overhead and limits the maximum achievable bandwidth when no shaping is used; see Figure 8.1. Figure 8.5 shows achievable bandwidth with encapsulation disabled, which results in much higher achievable aggregate bandwidths.

8.1.4 OpenVZ Containers Results

Figure 8.6 shows the aggregate bandwidth achieved as the number of client-server pairs increases. The maximum bandwidth of the OpenVZ architecture on the test machine was approximately 1.5 Gbps, occurring for two unshaped virtual links. However, for shaped links, the maximum observed bandwidth occurred near 1.2 Gbps, and when the number of pairs is large, this is an overall cap. There is also a clear diminishing trend of aggregate bandwidths as the number

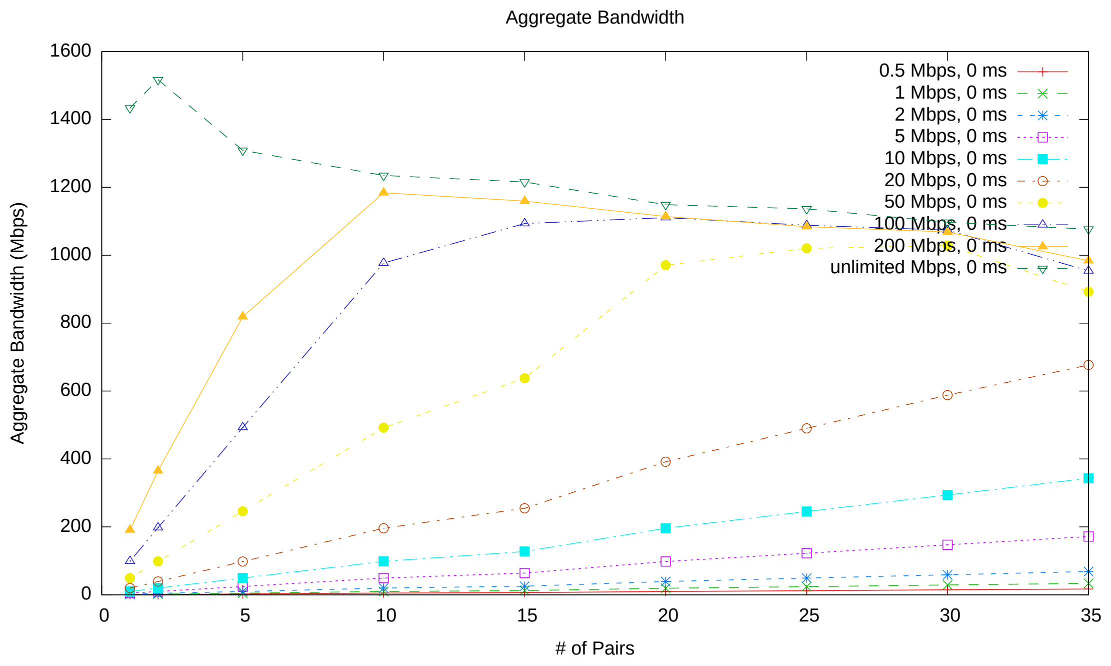

Figure 8.6. (NEW) OpenVZ: Aggregate bandwidth across all virtual links as a function of bandwidth shaping values and number of client-server pairs

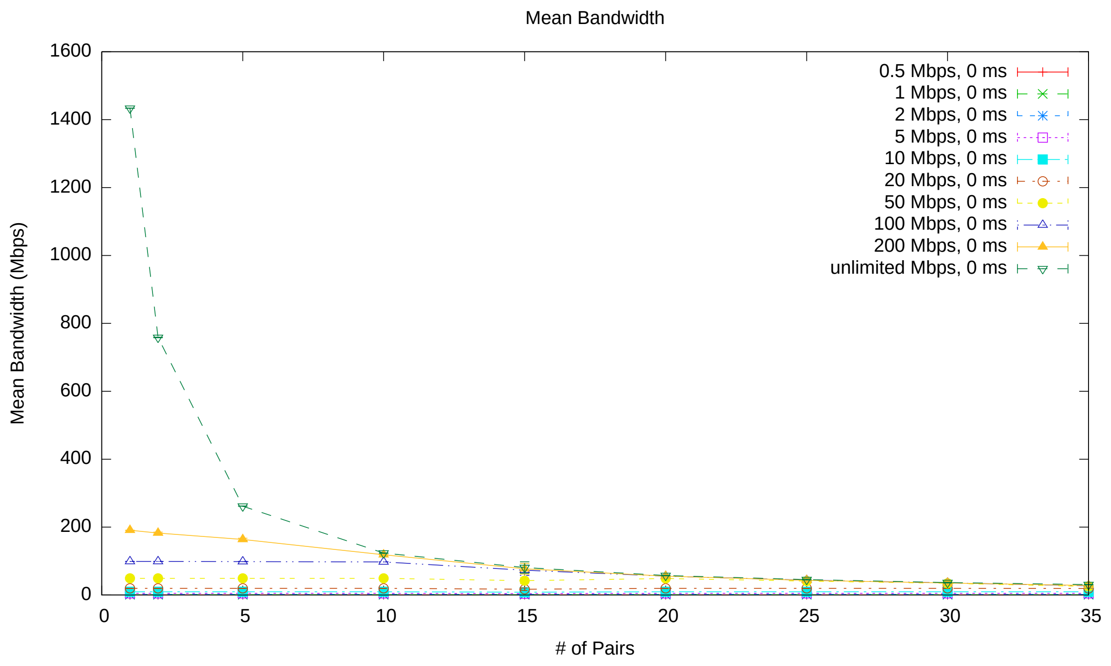

Figure 8.7. (NEW) OpenVZ: Mean bandwidth of each virtual link as a function of bandwidth shaping values and number of client-server pairs

of virtual links increases; we expect this trend to continue due to increased locking contention and general CPU overhead in the traffic shaping, and perhaps eventually to become limited by physical resource contention. Finally, there is a clear artifact appearing in the 15-pair results. In the OpenVZ testing, each per-pair subset of tests (i.e., parameterized with different bandwidths) were conducted on a single physical node, so there could be a hardware issue with the node that hosted the 15-pair tests. If it is not a hardware issue, it is possible (although unlikely) that 15 virtual links connecting 30 virtual nodes is triggering strange kernel behavior due to some special kernel constant. Regardless, the curves representing bandwidths less than 50 Mbps continue linearly aside from the dip at 15 pairs.

To judge fidelity more easily, Figure 8.7 shows the mean bandwidth achieved by each pair. At speeds below 20 Mbps per pair, fidelity remains high in all circumstances. At 100 Mbps, fidelity breaks down when there are more than 10 pairs.

Both overall CPU utilization (Figure 8.9) and kernel CPU utilization (Figure 8.8) grow as an ill-defined function of bandwidth and the number of pairs. At high bandwidths, relatively few pairs saturate the CPU. This saturation is one of the principal reasons why fidelity decreases as the number of virtual containers increases.

However, there are two problematic features of these graphs. First, in the case of a single unshaped virtual link, aggregate bandwidth is smaller than for two unshaped links, *but* mean kernel and overall CPU usage clearly establish that the CPU is nearly 25% idle. We will need to perform a much more exhaustive performance characterization of the network stack to establish the exact cause, but it is possibly related to TCP window size interactions on a high-capacity (unlimited) link with essentially no delay.

Second, the bandwidth curves greater than and including 50 Mbps reach nearly full CPU utilization in extremely sudden exponential jumps. These could be a result of locking contention in a number of areas exacerbated by high-speed 1500-byte packet flows, but we must understand these seemingly sudden increases better.

Memory utilization also increases dramatically as the number of containers increases (Figure 8.10). This increase in memory is almost entirely inside of the kernel because no memory-intensive applications were run. The decrease in memory availability is uniform across limited-bandwidth virtual links, which signifies that memory usage increases are due to increasing numbers of containers. This may be occurring because OpenVZ does not provide any block-level copy-on-write support in its virtual filesystem; hence, the kernel buffer cache may be duplicating work for each container despite the fact that many blocks are identical across containers.

Finally, we examined the interrupt rate (Figure 8.11) and internal kernel memory utilization as revealed by the slab cache responsible for storing fast-cloned `sk_buffs` (Figure 8.12). An `sk_buff`—socket buffer—contains per-packet headers, metadata, a protocol-specific storage buffer, and a pointer to the actual data contents; it is the primary vehicle for conveying packets through the network stack in the Linux kernel. These results have not provided additional insight into the questionable behaviors we discussed above; for instance, interrupts do not seem to become a clear bottleneck, and the slab cache for cloned `sk_buffs` seems able to grow (although the behavior of the 50 Mbps curve above 20 pairs is somewhat mystifying in itself).

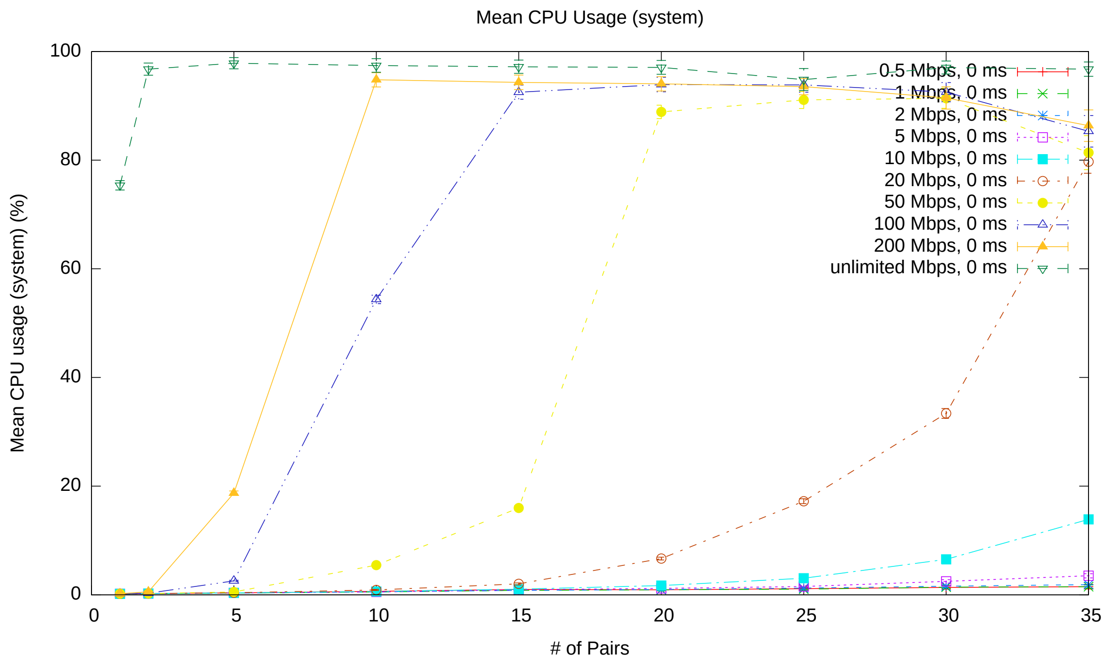

Figure 8.8. (NEW) OpenVZ: Mean percentage of CPU used by the Linux kernel as a function of bandwidth shaping values and number of client-server pairs

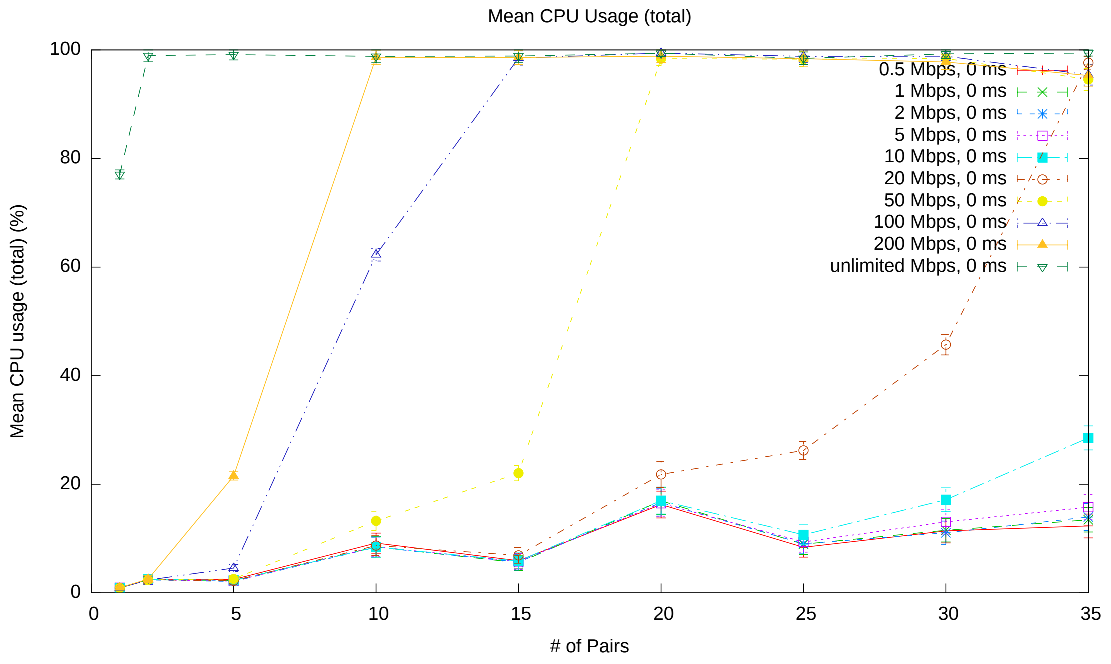

Figure 8.9. (NEW) OpenVZ: Mean percentage of CPU used in all contexts as a function of bandwidth shaping values and number of client-server pairs

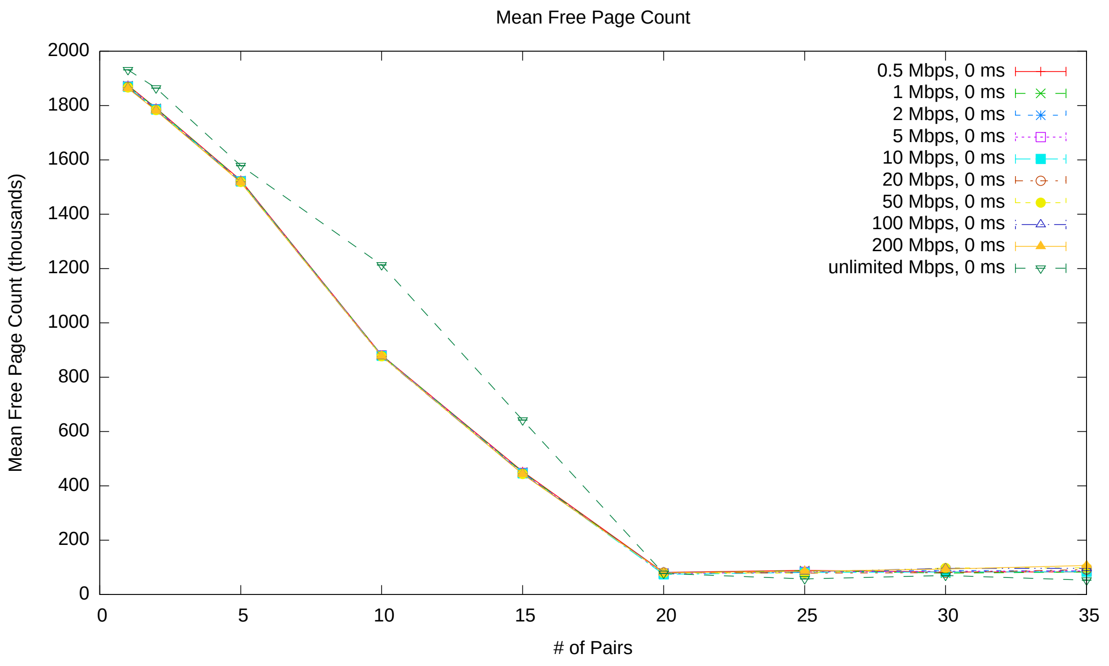

Figure 8.10. (NEW) OpenVZ: Mean physical memory free as a function of bandwidth shaping values and number of client-server pairs

8.1.5 Improving OpenVZ Scalability

Clearly, there is room to improve the performance of Emulab's OpenVZ support. Perhaps most importantly, we must conduct much more detailed kernel profiling to establish where in the network stack blocking is occuring, and why CPU can so suddenly become a bottleneck. There are likely opportunities to optimize traffic shaping that can help reduce contention, but we do not expect these optimizations to be of significant advantage. Primarily, we are interested in increasing the total available aggregate bandwidth so we can pack more virtual nodes and links onto physical hosts without loss of fidelity.

Although the results above did not reveal this, we know from experience that initial container creation is a large bottleneck. Since OpenVZ's virtual filesystem does not support a copy-on-write mechanism, there is no way to share identical disk blocks between containers. Moreover, this will negatively impact the size of the buffer cache needlessly, as was implied in the results above. Short of adding our own copy-on-write mechanism to the OpenVZ virtual filesystem code, there are at least two potential alternatives. First, we may be able to leverage the Linux LVM (Logical Volume Manager) block-level copy-on-write. Essentially, LVM provides support for creating a logical volume atop one or more physical devices, formatting the logical volume with a filesystem, and populating it. An administrator can then snapshot the logical volume into a new logical volume, which can be mounted and accessed using copy-on-write mechanisms. Second, Linux *vservers*, another container-based OS-level virtualization technology, implement

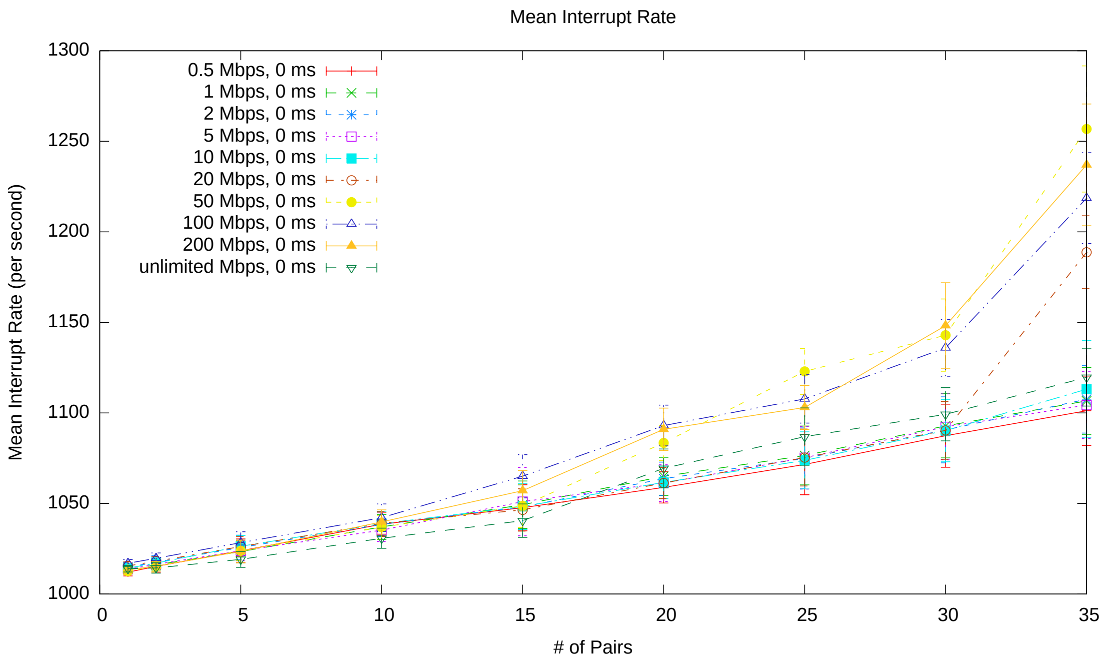

Figure 8.11. (NEW) OpenVZ: Mean number of interrupts per second when traffic is sent between different numbers of client-server pairs

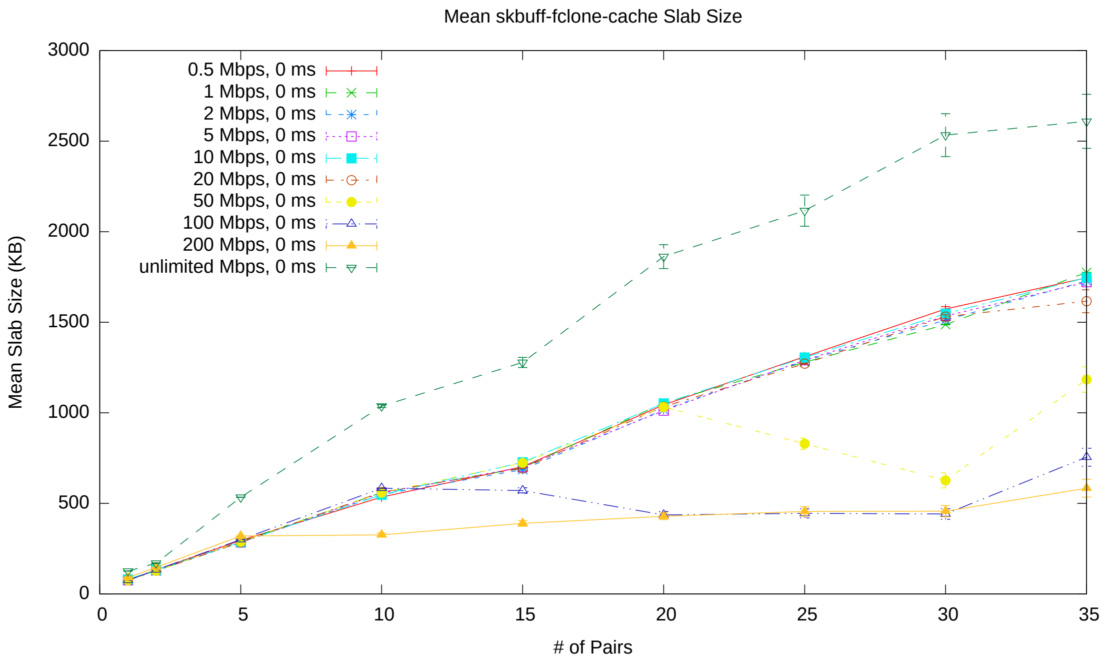

Figure 8.12. (NEW) OpenVZ: Mean memory consumption of the skbuff_fclone_cache slab

filesystem-level copy-on-write based on file hashing and hardlinks; it may be possible to port some of this code to the OpenVZ virtual filesystem.

OpenVZ provides a *virtual ethernet device* (a *veth*) that is a software device that manifests in both the root context and in one container. It provides a Layer 2 device that can send packets into the root context, where they may be routed elsewhere or bridged to physical devices—or to other veths. In Emulab, to build a virtual link between two virtual nodes residing on the same physical host, we bridge the root context halves of two veths together. However, by slightly modifying the veth driver with some additional metadata to construct direct paths between root context halves of two veth devices, we can eliminate all bridging overhead. Although this modification likely should not be used to create virtual LANs (since additional code that duplicates true bridging functionality would need to be added), we may be able to reduce some of the overhead from unnecessary bridging functions; the extent to which this might increase packet throughput is unknown at this time.

8.2 Future Support: Xen and VMware

Unfortunately, Emulab's Xen support is in its infancy, and we were unable to test it for this report. (We have no concrete plan to provide VMware support in the near future, either.) However, although we have not conducted any performance characterization tests as we have done for FreeBSD jails and Linux OpenVZ containers, there are still several obvious resource scalability issues that generally apply to heavyweight paravirtualization techniques that we discuss in this section. Since these techniques provide an emulated machine interface to guest operating systems, resource requirements are significantly higher than for the OS-level virtualization mechanisms already supported by Emulab.

The brute-force way to achieve scalability for heavyweight virtualization is to provision the range with high-end servers (e.g., 8-way quad core CPUs, 64 GB RAM, as many physical hard disks distributed atop as many I/O busses as possible). However, when experimenters are running non-virtualized tests, this will lead to significant under-utilization of resources as most tests will not require anywhere near the physical resources available on one of these nodes. Consequently, it is worth discussing current technology that may help share virtual resources more effectively to minimize physical host node requirements.

8.2.1 Sharing Memory Pages Amongst Virtual Machines

In paravirtualization environments, physical memory is partitioned between virtual machines. Such "hard" partitioning of memory can lead to significant fragmentation and thus inefficient use of physical memory resources. While virtual memory paging techniques can be used to address this, the well-known performance penalties associated with paging, coupled with the already significant overhead of operating inside an emulated machine, make avoiding memory swapping to disk important. Consequently, it is worthwhile to consider techniques that reduce per-virtual machine memory consumption by sharing virtual pages or their contents, between virtual machines. Of course, this must be done securely—only identical content can be shared, and then only in a read-only, copy-on-write style, so that when each VM attempts writing to a shared page, they are given their own copy that is unshared.

Recent research in this area ([21]) implements page sharing, subpage sharing, and memory compression within the Xen VMM, with reasonable overhead, and claimed factors of 1.5–2.5 increase in memory savings over VMware ESX server. It may be possible to leverage these or similar techniques in some experiments, especially those with similar guest operating systems and workloads, to reduce per-VM memory requirements, but more testing at larger scales would be required to establish just how helpful this could be.

8.2.2 Increasing Virtual Disk Speed

Hard disks are already a performance bottleneck in current PC architecture, and this problem will only be exacerbated on a physical node hosting many paravirtualized virtual nodes. We envision several techniques that may help distribute virtual disk accesses more effectively across physical disks. As we discussed in the context of OpenVZ improvements, LVM's block-level copy-on-write mechanism is of obvious utility here to minimize disk footprint for similar virtual machines. However, this by itself does not provide a speedup, aside from the fact that the root context buffer cache is not duplicating effort, and similar read access patterns amongst the virtual machines may result in fast effective reads.

Virtual disk write speed is a major unresolved concern. Since it is unlikely that nodes would be over-provisioned with a single physical disk for each possible virtual node that a given physical node type is expected to host (since these node types would then potentially be over-provisioned for non-virtualized experiments). We need to investigate and determine effective mechanisms to stripe writes across multiple physical disks. It is not clear that simple RAID will help here, since the effects of striped writes must be evenly balanced across all virtual disks to achieve maximum multiplexed write speed. Perhaps Linux LVM (or a modified version of it) could work cooperatively with physical RAID controllers to achieve such a balance—but these are preliminary thoughts, and require further investigation and testing once NCR virtualization requirements are more defined.

Alternatively, an NCR facility might include a dedicated storage area network allowing high-throughput for a large number of virtual machines to a central well-provisioned storage server. The expense of the storage server and SAN is amortized over a large number of nodes, possibly making it a more attractive option than node-local disks.

8.2.3 Network: Fast Virtual Links

Just as in the lightweight OpenVZ mechanism we discussed earlier, there may be opportunities to "directly connect" two paravirtualized virtual machines via a virtual link more directly than bridging root context veth halves in the hypervisor (or, for instance, in Xen's case, in dom0) to form the link. There has been much research, both historically and currently, on fast intra-node IPC mechanisms which might be amenable as a bridge replacement—for instance, allowing direct transfers between VMs. There are obvious security implications, but this or a similar option may be worth exploring.

8.2.4 Leveraging Architectural Improvements

Intel and AMD have virtualization support in hardware, enabling VM frameworks to be more efficient. Xen and VMware among others take advantage of this technology, including support for dedicating physical processing cores to individual VMs. Further, Intel's VT-d I/O virtualization technology [29] introduces hardware-level DMA and interrupt remapping allowing for dedicating IO devices to VMs. Emulab can take advantage of these technologies in current and future facilities that include multi-core processors and multiple network interface cards.

8.3 Improving Resource Configuration and Runtime Service Scalability for Virtual Nodes

Regardless of the specific virtualization technology used, there are many common scaling issues that must be addressed related to configuration and control of virtual nodes.

In Emulab today, virtual node setup on a physical host is serialized to avoid resource overload during the often CPU and I/O intensive initialization phase. While not a significant overhead when setting up FreeBSD jail-based vnodes, which require on the order of 30 seconds each to setup, it will become an increasingly significant burden as virtual node setup times increase. The solution is straightforward: allowing for parallel setup of virtual nodes as constrained by available node resources. We have began implementing this in Emulab.

As seen throughout this chapter, many of Emulab's control processes are currently "flat," with nodes interacting directly with the central "boss" server, and will require the introduction of hierarchy to distribute the load as Emulab scales up. Virtual nodes present an additional opportunity for such hierarchy: using the physical host as caching proxies for both configuration and runtime data. Of particular interest here is the ability to not only cache commonly used disk images, but also to use them as "roots" of copy-on-write virtual disks for hosted vnodes. We currently use this technique in our prototype Xen support and plan to extend support for it.

As discussed in Section 11.3.1, Emulab physical vnode hosts are dedicated to, and completely accessible by, a single experiment. The latter is problematic as it means that vnode hosts must be completely cleansed between uses. This limits out ability to cache largely static, range-wide state (e.g., generic VM images or other repositories). If such caches persist across uses, setup times can be greatly decreased, thereby increasing the practicality of large-scale vnode tests. We have taken the first steps in this direction—removing direct experiment access to physical hosts– as a result of our recent support for sharing physical vnode hosts between experiments. On this we will begin to build up persistent caching mechanisms.

Finally, Emulab can perform smarter allocation of physical host resources to virtual nodes. Taking advantage of underlying architectural virtualization improvements, as described in Section 8.2.4 is one way. However, even in their absence, Emulab can take advantage of available hardware redundancy; for example, dedicating, or at least distributing the load across, multiple physical disks. By better utilizing physical node resources, we can increase our ability to colocate VMs and thus provide better scaling.