SAND2010-2078C

Ten Million and One Penguins, or, Lessons Learned from booting millions of virtual machines on HPC systems

Ronald G. Minnich[*]
Sandia National Laboratories
rminnich@sandia.gov

Don W. Rudish
Sandia National Laboratories
dwrudis@sandia.gov

Abstract

In this paper we describe Megatux, a set of tools we are developing for rapid provisioning of millions of virtual machines and controlling and monitoring them, as well as what we've learned from booting one million Linux virtual machines on the Thunderbird (4660 nodes) and Hyperion (1024 nodes) clusters. As might be expected, our tools use hierarchical structures. In contrast to existing HPC systems, our tools do not require perfect hardware; that all systems be booted at the same time; and static configuration files that define the role of each node.

While we believe these tools will be useful for future HPC systems, we are using them today to construct botnets. Botnets have been in the news recently, as discoveries of their scale (millions of infected machines for even a single botnet) and their reach (global) and their impact on organizations (devastating in financial costs and time lost to recovery) have become more apparent. A distinguishing feature of botnets is their emergent behavior: fairly simple operational rule sets can result in behavior that cannot be predicted. In general, there is no reducible understanding of how a large network will behave ahead of "running it". "Running it" means observing the actual network in operation or simulating/emulating it[1]. Unfortunately, this behavior is only seen at scale, i.e. when at minimum 10s of thousands of machines are infected. To add to the problem, botnets typically change at least 11% of the machines they are using in any given week, and this changing population is an integral part of their behavior.

The use of virtual machines to assist in the forensics of malware is not new to the cyber security world. Reverse engineering techniques often use virtual machines in combination with code debuggers. Nevertheless, this task largely remains a manual process to get past code obfuscation and is inherently slow. As part of our cyber security work at Sandia National Laboratories, we are striving to understand the global network behavior of botnets. We are planning to take existing botnets, as found in the wild, and run them on HPC systems. We have turned to HPC systems to support the creation and operation of millions of Linux virtual machines as a means of observing the interaction of the botnet and other non-infected hosts.

We started out using traditional HPC tools, but these tools are designed for a much smaller scale, typically topping out at one to ten thousand machines. HPC programming libraries and tools also assume complete connectivity between all nodes, with the attendant configuration files and data structures to match; this assumption holds up very poorly on systems with millions of nodes.

[*] Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energys National Nuclear Security Administration under contract DEAC0494AL85000. SAND- 2010-0986C.

1. Introduction

As this is written, the discovery of a global botnet has been reported that spans thousands of organizations. We used to be surprised when thousands of machines were "owned", or taken over. That surprise now seems quaint: the number of owned machines is now routinely measured in the millions. The systems span the globe, and are structured and function without regard to physical, organizational, or network boundaries. Botnets exploit communication channels as varied as IRC and bug trackers. They are flexible: changing 11% of their members each week, or around 5000 per hour. They are resilient, to the point that one botnet was recently reported as existing on its own, without a controller. They tolerate faults. They frustrate attempts to interact with them, including autonomously launching DDOS attacks under certain circumstances[6]. And, finally, they are available: construction kits are offered, in the right corners of the Internet, for as little as a few hundred dollars[12].

In fact, we computer scientists might feel a bit of embarrassment: botnets implement all the grand goals of the grid, but they were not written by any of us – or, at least, anyone who is admitting it: there is reason to believe that much of this development effort is underwritten by criminal organizations.

Botnet binaries, or payloads, are designed to frustrate traditional reverse engineering, using automated code obfuscation tools that wrap the binary in a bundle of spaghetti code. But the real problem is much more subtle: botnets, once they come into existence, and reach a certain scale, exhibit emergent behavior: their actions cannot be understood by examining one bot, but rather must be observed as the collective behavior unfolds.

Defending against botnets requires more understanding than we have today. Understanding them must be preceded by observation of their behavior. At Sandia, we have begun a project to gain such understanding by booting millions of virtual machines, installing a botnet, and monitoring what it does and how it interacts with other servers in the network. We are, in short, building an emulated internet in an HPC system. The scale of the HPC system, multiplied by the scale of the large number of Virtual Machines (VMs) we can run per node, gives us the scale we need.

To make the emulation realistic, we need a minimum structure in the network. At minimum, we need to provide the set of services that botnets require, such as DNS and BGP; and we will also need components that botnets are known to use, such as IRC, web, and mail servers. We must also deal with frequent node reboots; in fact, we will probably have tasks that force near-continuous reboots of VMs at random, to more accurately model the real world.

It is also important that the virtual machines be very lightweight: after all, we need to run at least 1024 VMs per physical machine. Hence, we need very lightweight Linux images, as we are limited by memory. The images must also, in turn, not consume a lot of compute power beyond that needed to emulate infected and uninfected machines; we are over-committing the CPU by a factor of 1000. We might not run a full Windows desktop, for example, but rather just enough of one to "close the loop" for a bot to infect it. Finally, the VM image should minimize extraneous activity, a.k.a. "OS noise", so it can minimize work it does that is not useful for the overall simulation.

Here we can see a relationship to HPC. The host controlling operating system and the VMs need to have a small memory footprint. This requirement applies to HPC nodes as well, since the application needs most of the memory. The VMs need to use as little compute power as possible, so as to leave the CPU available for the application; and, as a final point, the VM image must minimize noise in order to attain the highest possible efficiency. We have begun to suspect that lessons we learn from this research will apply to future exascale machines.

Configuring millions of VMs, done the traditional HPC way, would be a daunting task. HPC systems today are designed with a few key – and, in our view, unrealistic – assumptions in mind. For example, it is a given that the machine is all up or all down – that it all boots at once, and when one node is lost, the whole machine must be restarted. As anyone who has used or administered these systems knows, they do not deal well with partial reboots.

The "perfect reliability" assumption pervades most HPC software and hardware. For example, on many HPC systems, there is hardware-provided reliable transport. A lost packet on these machines is as significant an event as a memory error. Unwinding the network interface when this event occurs is hard, and it is usually simpler to reboot. On Infiniband, there is a single point of failure called the OpenSM. Remote DMA does not continuous handle failures gracefully, to say the least.

The configuration software is equally unrealistic. Some Linux-based HPC systems, even those with tens of thousands of nodes, require files such as DHCP setup files, which have an enumeration of a set of values for *every* node. In fact almost all Linux HPC systems we have looked at, even large systems such as the Cray XT series, have files or kernel structures that are linear in size to the number of nodes, e.g. on the XT series there is an ARP table entry on each of 30,000 hosts for the other 29,999 other hosts; this on a machine which is not using an Ethernet internally, and on which a simple polynomial suffices to map the nodes X,Y,Z coordinates to an IP address.

Reliability, Availability, and Service (RAS) systems don't provide enough data. The sampling rates are far too low: 10 minutes is a typical number. A botnet with a million components will have changed 1000 of them in that time, and will have sent many messages. We will not be able to see any texture in the system statistics. But even systems with such a low sample rate can't simply be ramped up: one system we know of, when the sample rate was increased to only once per minute, ended up consuming 60% of the CPU. We are targeting a sample rate of once per second for all ten million VMs.

Here are some representative numbers for systems of ten million nodes, configured with existing HPC tools.

1. DHCP file: several gigabytes (whether distributed or not, it takes this much space)
2. ARP table: 100 megabytes
3. monitoring: 10 million samples/second, around 128 bytes per sample

Finally, the programming models don't scale either. Programmers take it as a given that their applications can have cognition of every node in use, can control placement, and that in some manner they can control each and every node. That may work to small node numbers found on a BG/P system, with only 40,000 nodes; it is completely impractical for ten million nodes. While, e.g., MPI is commonly used as a "process scaffolding" even on many-task applications at a small scale[4], it cannot be used at our scale.

All of the assumptions underlying HPC software have been found wanting for an environment in which we boot millions of machines. Hence, we have been challenging these assumptions, and building new models for multi-million OS environments. In the process, we have begun to believe we may be building systems software environments for future multi-million-node clusters: the botnet emulation support system can be a prototype system for future HPC machines. Botnet concepts may provide key insights for how we structure that software so it is resilient, scalable, and yet still controllable.

2. Work to date

The work to date has proceeded in four phases. The first phase did not involve botnets at all. In this phase, we used the Lguest virtual machine[9] to boot a one hundred node virtual cluster on an IBM X60 laptop. Once we had created lightweight RAM disks for the guests, we had 8M images and could boot all 100 in 30 seconds. This virtual cluster was used to refine the XCPU[8] software as well as extensions to oneSIS to support the lightweight RAM disks for the VMs. It was at this point that we realized we could bring this technology to HPC systems; rather than booting 100 machines on a laptop, we could boot 1000 machines on 1000 HPC nodes. The application to botnets struck us as a useful first demonstration.

The next step was botnet-oriented. We further extended the oneSIS software to allow us to boot 5000 guests on a 70-node cluster. We continued to use Lguest in this instance as the CPUs were too old to allow us to use hardware virtualization. We successfully modeled the propagation of a trivial worm (modeled after the 1989 Morris worm) in this 5000-node environment.

The third step was our first million-node run. We performed this work on the 4600-node Thunderbird cluster at Sandia. We extended the oneSIS software to support 250 nodes per node, We crashed a number of the support systems on Thunderbird while getting this many VMs to boot: the Ethernet switch could not handle one million MAC addresses at one time and basically went into a seizure as it frantically tried to shuffle its overloaded associative tables. We were grateful for this problem, as it neatly sidestepped the issues that came with one million ARP table entries per node. The Infiniband network was intolerant of our frequent reboots, and in the end we did not use it. To allow inter-VM communication across the physical network, we partitioned the IP address space on a per-physical-node basis and installed 4600 static routes on each of the 4600 nodes. The final problems all involved the RAS and control subsystems, particularly IPMI. IPMI provides an unreliable serial console with 1200 baud throughput. If anything goes wrong, the sysadmin has to run out and hit many reset buttons – 512 in one case. IPMI is not usable for management of clusters where nodes are rebooting frequently.

Starting commands on this scale proved difficult. We started out using XCPU, but had trouble getting the communications to scale. That was disappointing as XCPU, in its more recent incarnations, had a tree spawn capability that should have fanned out well. We fell back on a hack: we used pdsh to start up command files on the physical nodes that, in turn, started up 1000 processes on the VMs on each node. We thus created a ”scalable” process startup, but it is hardly what we want.

Managing the output of one million processes is also a challenge. There is a clear need for intermediate levels of filtering, combination, and control, which in turn implies a hierarchy. We did experiment with some of the existing software that is designed with hierarchy in mind, such as Supermon[11] and TBON-FS[3]. Supermon was only designed to for a tree-based data collection, not processing, and is not appropriate for this purpose. Supermon also uses a ”pull” model, in which data collection is initiated by a request from a central collector. On multi-thousand node systems this had scaled extremely well, far better than existing ”push”-based models such as Ganglia: Supermon can sample thousands of nodes at several hz., whereas Ganglia, on the same scale is limited to 1/600 hz. Nevertheless, the pull model did not hold up well at one million nodes. TBON-FS was promising but did not work well in our testing. As a result we are moving to a modified push model called Pushmon.

Overall, the inability of HPC software to scale is what ultimately presented problems for million-node runs. The tools, usable for several thousand nodes, were completely unusable for millions of nodes. At one point we ran into a piece of code that had a hard limit identifier named UNLIMITED set to 600,000 with a comment stating that no one would ever have a 600,000 node cluster.

3. Software we are developing

The software we are developing is designed to run with only one configuration file, and without having any information about any single node, but rather about node ranges. Further, programs that need configuration are being written to determine that information from the environment; we call this technique computational configuration, as compared to file-based configuration. The programs examine their environment and, using simple rules, determine what their configuration should be. Since computational configuration is embarrassingly parallel it can be very fast.

3.1 VMatic

VMatic is a tool for rapidly provisioning virtual machine environments. It is an extension of oneSIS, and has a similar configuration file, shown in Figure 1.

```
PHYSICAL_HOSTS:      ehyperion[2-37,74-145,148-283,286-429,
VMS_PER_HOST:        1024
START_VM_NETWORK_AT: 1.0.0.0
VM_NETMASK:          255.255.0.0
VM_MEM:              30

HOST_OVERLAY_DIR:   host_overlay
GUEST_OVERLAY_DIR:  guest_overlay

GUEST_INITRAMFS:    boot/initramfs-guest.img
HOST_INITRAMFS:     boot/initramfs-host.img

GUEST_KERNEL:       boot/bzImage-2.6.32.8.lguest

# Include modules in HOST image
USE_MODULES:        false
KERNEL_NAME:        2.6.31-17-generic
INC_MODULES:        e1000 tg3 kvm-intel kvm-amd

# Which VM type?
INC_LGUEST:         true
INC_KVM:            false

GUEST_IMAGE_DIR:    src/guest_image
HOST_IMAGE_DIR:     src/host_image

USE_BPROC:          true
OUTPUT_BPROC_CFG:   hyperion_bproc.cfg
BP_LISTEN_INTERFACE: br0

# Initramfs template
PATH_TO_TEMPLATE:   src/guest_image_template.tgz
```

Figure 1. Sample VMatic configuration file

This simple file is used to generate 1,008,640 fully networked virtual-machines on Hyperion, an HPC testbed cluster at the Lawrence Livermore National Laboratory. There are 985 physical compute nodes used in this experiment with 1024 virtual machines on each physical node. Each physical node maintains its own 16 bit network and serves as a Linux router and DNS server for the local VMs. A DNS server is used as opposed to a local /etc/hosts file since a fully populated host table with 1 million entries is over 50 megabytes and would exceed the memory allotted each VM. Communication between the local VM and the host OS is relatively fast since latency is almost zero and does not put any additional burden on the physical network. While we have found this topology to be the most scalable, we do have plans for supporting additional topologies, allowing users to define richer environments with virtual Cisco routers and different layouts to produce a more realistic Internet emulation.

There is an interesting issue with node numbering as seen in the PHYSICAL_HOSTS line. The compute node numbering on this machine is very irregular. It is driven by node locations in a rack, rather than by any sensible scheme. As a result, we are stuck with enumerating node ranges, because the node number can not be expressed with an equation. For a very large machine, this strange numbering would increase the size of the configuration file quite a bit. This situation shows the problems that ensue when system builders are not used to thinking on a large scale. We need to get the community thinking in terms of configuration that works computationally, as that is the only way we can work on a large scale.

VMatic produces system images that can be uploaded to compute nodes via network boot. The user is able to specify the configuration of both the host and guest images, and allows the user to set up additional included files via the HOST_OVERLAY_DIR and GUEST_OVERLAY_DIR options. Through the use of multiple VMatic configuration files, a user can maintain separate build configurations for a variety of experiments on multiple clusters. The main config file is used by a new command, `megatux`, to configure the kernels, ram disk files, and other attributes of the host and guest VMs. The end result is a bootable kernel and initramfs image that can be deployed via network boot. On an existing network boot cluster, the only change needed is a change to the boot target and a kexec or reboot of the cluster.

Once the physical host has done a DHCP request, it has all the information it needs to configure its own services and configure its local virtual machines. The MAC addresses, virtual Ethernet devices, and routes to other networks are all computed as a node starts up; there is no central store of MAC addresses for all the VMs. This is the beginning of computational configuration of most if not all other boot time parameters. A single virtual machine takes about one second to fully boot, less the time it takes for the Ethernet Bridge to enter a forwarding state. At this time, a user may run their predefined runtime scripts or issue commands to the VM via Xproc (described below).

In addition to provisioning HPC systems, VMatic can also be used to provision lightweight VM's on a local machine. This has proven to be invaluable for our development work as it gives each team member the ability to boot their own virtual cluster on their laptop. On our laptops, we can reconfigure and start 100 new VMs in less than 30 seconds. This speed makes testing easy. Developers now have an immediate, convenient, and reliable way for performing automated testing on their code sets without consuming precious time on limited HPC resources. This mechanism was used in the creation of our first botnet prototype with successful results when we started scaling out.

3.2 xproc

Xproc is the latest in a line of process startup systems we have developed, starting with BProc[5] and continuing with XCPU[8]. Xproc uses BProc's wire protocols and its I/O forwarding design. Xproc uses a tree spawn mechanism similar to that of XCPU, and also moves all the libraries a given command needs to run, as does XCPU. Instead of the ad-hoc command tree spawn technique that BProc uses, Xproc sets up a persistent tree of servers that reduces the tree spawn overhead. Finally, Xproc uses intermediate nodes in the tree to aggregate I/O from remote processes, instead of counting on the top-level command to aggregate I/O as in BProc.

Before we discuss xproc we first give an overview of bproc and xcpu. those familiar with the two systems can skip the next sections.

3.2.1 bproc

BProc provides a single unified /proc image of a cluster – hence its name: Beowulf /proc, or BProc. The key concept of BProc is the use of process-directed (i.e., voluntary) migration from a *master* node to a *slave* node. When a process migrates from a master, it leaves behind a *ghost*, which provides a hook for process operations such as kill, ptrace, and wait. The ghost has no virtual memory, and is little more than a symbolic link to a remote process.

BProc process migration and process operations are supported by a set of intrusive patches in the Linux kernel. A major operation is the migration itself, as it provides support for ”freezing” the process, bundling it up, and sending it to a bproc slave daemon. The slave deamon, in turns, supports a ”thawing” operation to restart the process. A process may further migrate from a slave node, the only difference being that it does not leave a ”ghost” behind: to keep the accounting correct as to the location of a remote process, the master node must participate in migrations from slave to slave. A process migrated from a master to a slave can also quickly replicate itself from one slave to others; this capability forms the basis of the bproc tree spawn. BProc systems such as the Los Alamos Pink cluster demonstrated an ability to start up a 16 MByte MPI process, across 2048 CPUs, in 3 seconds.

The kernel, master, and slave daemons form a triumvirate which manage the movement and control of processes. Users were provided with a process name space, presented on a single master node, which spanned all the processes in the cluster.

BProc presented node status and control via the BProc File System, BFS, in which each node was represented as a file. Extended attribute operations, as well as ownership and mode, could be set with standard Linux commands, and these settings in turn controlled access and node state.

BProc scales well to to a few thousand nodes. It does not scale to millions of nodes. A ps command which returns millions of lines of processes is barely managable. Having a file system with a file per node is impractical past a few thousand nodes. The system does not deal well with node outage, and the requirement that all slave node accesses be synchronized with the master node is clearly impractical. For scalability and reliability, the nodes must be decoupled.

3.2.2 xcpu

XCPU provides a file-oriented access model to a cluster process management system. Instead of the custom protocols of BProc, XCPU uses 9p[7] as the underlying protocol, with resources represented as file names. The tree is rooted with a set of directory nodes. In each of the node directories there is a set of process directories, one for each remote process. In each process directory there are files for the process code; standard input, output, and error; and miscellaneous files for controlling and debugging the process.

XCPU is designed for hierarchy in the manner in which it spawns processes. That said, XCPU suffers from the same scalabilty issues as bproc: there is one ’control point’ for each node and each process on each node, and each of these is represented in the file system namespace presented at the top level. A list of the name space representing the node resources would, by itself, run on for 10 million lines. This model is simply not scalable to one million systems. We made use of XCPU in the initial Megatux work but it quickly became apparent that we need to build a new and more scalable process framework.

3.2.3 Process models that scale

A process model that scales has several important attributes:

- Tree structure: the node address space, communications, and control must function in a hierarchical manner. No node should ever need to communicate with all other nodes – it is unlikely that they will all be up at the same time anyway.
- Ad-hoc: the creation of the hierarchy is dynamic, not controlled by a configuration file. Configuration information, such as node naming, must be described by an algorithm of polynomial, not a static list.
- Dynamic: the hierarchy is continuously changing in response to failures and node restarts.
- No specialization: nodes must be able to perform any role. A node that is functioning as a compute node must be able to take on the role of manager of other nodes on demand.
- Aggregation model: Individual nodes are not visible; applications operate in terms of groups of nodes.
- The fate of an individual node is unimportant (we make one exception: the root node).
- Decoupled (or asynchronous) operation: the application can inititate an operation (e.g. start a program on a set of nodes) but can not assume that the operation completes successfully.

3.2.4 xproc

Xproc is designed around the principles outlined above. XProc is designed from the start for hierarchy. XProc configuration files are being modified to contain near-executable code in the form of equations that define the mapping of node Ip addresses to node names. In contrast to the BProc master/slave structure, with two specialized daemons, the XProc daemons are the same everywhere and can take on the master or slave role as needed. XProc is designed to deal with nodes as aggregates, and as part of our design we have removed all the aspects of BPRoc that were designed to work per-node, e.g. the BProc file system. Nodes, even the root node, can fail and the rest of the nodes will reconnect around the fault, without much fuss; in the original BProc, loss of the master daemon would take down all slave nodes associated with that master, hardly a resilient structure.

An xproc process tree consists of a root server, intermediate daemons playing both roles, and a set of daemons at the leaves. The ultimate clients are at the leaves of the tree, i.e. individual processes. The ultimate server is at the root of the tree, i.e. the program that initiates the million or more commands. If the root program is lost, it must be restarted ,but the state of the other daemons is affected only to the extent that they must reconnect and rebuild the tree – running applications are not lost. Daemons in internal nodes of the tree control processes below them, and relay data up and down the tree.

Xproc startup uses a dynamic configuration. At the beginning there is one master. Slave nodes contact the master using the existing BProc protocol. Consider the case where there are N total nodes. The first $\sqrt{N}$ slaves to contact the master are designated as secondary masters. Subsequent slaves that check in will be told to check in to the secondary masters instead, a process we call redirection. The root can ensure a balance of slaves to secondary masters by using a simple round-robin allocator. The process is continuous: slaves that lose connection to their master can go to the root to be reassigned. The only piece of information the slaves need to have to boot is the IP address of the master. The process is also recursive, a change from the Bproc single-level tree model.

Key to making this system work in the presence of failure is to minimize the amount of state retained at all levels. Given the rate of failure, more retained stated equates to more state that needs to be unwound when failures occur. It is best to retain not state at all. The root node retains no information about redirection commands to other nodes.

Each slave is a master of all the VM guests on its node. Hence, the process repeats for the VMs on each physical node.

For physical (non-virtual) nodes, xproc uses the bproc technique of pushing the binary out to the node. We improve on the bproc code with some extensions from XCPU. The bpsh command determines all the libraries a program needs and builds an in-memory cpio archive of the binary and its libraries. We further allow the user to select additional directories and files to send with the binary. It is thus quite easy for the user to send a set of binaries and input files to a node for execution.

At the node, when the command and its cpio archive is received, the daemon creates a process-private mount at `/xproc`, and unpacks the files there. As long as the command and its children are executing, the process-private mounts is available; once the process and its children have left, the mount disappears. The process-private mount ensures privacy between multiple users on the same node and eases administration. Note that users can still leave persistent files in share ddirectories such as /tmp.

We further allow users to specify that a local executable can be used, as in XCPU. For this case, the cpio archive can be zero-length. Because the basic onesis initrd includes a useful set of binaries, this process is convenient as well as very fast. We have hence relaxed the BProc/XCPU model of always importing a program.

For host to guest VM commands, it makes no sense to send 1024 copies of a binary to VMs running on the same machine. We use the private mount point to advantage here: instead of mounting a RAM disk, we mount a shared block device that is written by the host, and read-only to the guest. We then invoke the local-execution switch: since the binaries are by any reasonable definiton local. While there is an issue with shared block devices, we neatly finesse it by mounting the device read-only; other potential issues are resolved by the fact that the mount goes away when the command exits. The result is a very efficient system for physical machine-to-machine process startup, and a system that is near-optimal for host to guest process startup.

3.3 pushmon

Pushmon is a hierarchical monitoring system built from Supermon[11]. Like Supermon, Pushmon uses S-expressions to describe the data, and is designed for hierarchy, with Pushmon nodes functioning as both clients and servers. Unlike Supermon, Pushmon relies on a push model, with data being periodically pushed from the leaves to the root. Pushmon is also self-configuring, with the nodes using a low-cost computation to determine where their parent in the tree is, up to the root. Finally, Pushmon is designed not to just group S-expressions together, as Supermon does, but also to perform computations on the S-expressions so as to reduce the data load on the network. The computations to be performed can themselves be defined by S-expressions, and interpreted, allowing a great deal of flexibility, up to and including symbolic computing. See Figure 2.

```
(("MARK: 1266084142.710273")0x4336 "o0x4336 s1 #0")
(("MARK: 1266084143.272552")0x924e "o0x924e s1 #0")
(("MARK: 1266084145.387336")0x9879 "o0x9879 s1 #0")
```

Figure 2. Sample of Pushmon output. MARK: denotes the Unix epoch time from the root node and is embedded with the original messages using S-expressions.

Data load on the network is also reduced when the VM's relationship to their host OS is taken into account. When considering the fast communication path between a VM and its host OS, Pushmon can be used as an effective aggregator to collect messages from their child VMs before pushing to the root minimizing load on the physical network.

We are working to build an efficient virtio[10] transport for guest to host Pushmon communications. In spite of the plethora of virtio software that has been written, there is nothing that resembles an efficient pipe. We plan to remedy this problem.

4. Things we've learned

4.1 Which VM?

Over the past few years we've worked with a number of virtual machine systems on Linux, deploying them in various modes, from standalone USB boot media, to virtual nodes on a laptop, to tens of thousands to a million nodes on an HPC system. We've learned a bit about the strengths and weaknesses of each one.

4.2 Xen

Our earliest work was with Xen[2] 2.0 and 3.0. We ported Plan 9 to both these systems, in the process learning much about their structure. We found that Xen performance was fairly good, especially in 3.0; the system was carefully crafted for performance. Embedding drivers in the host Linux kernel, instead of in an external process, provided better throughput: kernel builds on Plan 9 ran in 12 seconds on Xen and over 70 seconds on early releases of KVM.

Xen is not without its issues. The Python management framework, with its XML-RPC, proved to be difficult to set up in a lightweight mode on a USB stick; these problems carried over to setting up lightweight VMs. Xen needs a huge amount of base software just to work at all. The hypervisor/host kernel split requires using a multi-boot loader, which limits options on network boot. Finally, the performance advantage is not as great as it used to be. We tried using Xen for Megatux but in the end found it to be too fragile when used outside a full Linux desktop. It has too many dependencies on a fully configured environment.

4.3 Lguest

Lguest[9] is a ”paravirtual only” hypervisor. It does not support real I/O devices, only emulated ones. There is not an external process; the kernel memory space forms part of the overall lguest process. The kernel driver switches between the kernel, running in Ring1, and the support program, running in Ring 3. Communications between the two is via virtio[10] queues.

Lguest performance is in some ways stunning, with a startup time for a kernel of roughly a second. Lguest I/O has never been particularly fast for I/O, however, with Plan 9 kernel builds running at almost 120 seconds the most recent time we measured it. Lguest is limited to 32-bit mode and therefore presents a problem for application requiring a x86_64 architecture.

Because of the integration of the guest VM and the support process in one address space, Lguest is very easy to use in a limited environment. We've built an Lguest-based distribution called THNX which boots to a simple shell with a BusyBox environment and supports a Plan 9 guest. It works well and is easy to use.

On newer systems, we can boot 1000 Lguest virtual machines in under a minute. The load on the host, once these machines are running, is not measurable when the machines are idle.

4.4 KVM

KVM is a newer hypervisor that functions only on processors with virtualization hardware, such as Intel VT or AMD SVM. KVM integrates a hypervisor into the kernel directly, avoiding the hypervisor/kernel split of Xen. KVM supports VM guest IO using QEMU, i.e. an external process.

KVM is a bit harder than Lguest to integrate into a lightweight VM image, due to its dependency on QEMU, which is a rather heavy program. QEMU depends on no less than 52 shared libraries, requiring a total of 14 Mbytes of space. While this space is not required in each guest, and the code is shared, the data is not. The virtual address space of each QEMU is 60 Mbytes, with a resident set size of at least 8 Mbytes.

We were able to measure KVM performance on newer systems. On AMD Barcelona systems with 32 Gbytes of memory we could boot 1000 virtual machines in under 2 minutes. Even on an X300 laptop we can boot 50 VMs with no trouble. The machine is completely unusable until the guests are started, at which point the overhead is very low so long as the machines are idle.

4.5 The VM we chose

Xen was never in the running, due to its complex runtime support requirements. The choice came down to KVM or Lguest. We expected that on new systems with hardware virtualization support, KVM would be the clear winner. Much to our surprise, the best system for our uses is Lguest on 32-bit hardware with Paging Address Extensions (PAE) enabled. PAE allows 32-bit mode kernels to use 64 Gbytes of memory, which is more than we have on any cluster node. The footprint of our Lguest guest was 20M virtual with a 12MB resident set size. The entire lguest guest and user mode support was barely larger than the KVM QEMU user mode support by itself.

5. Experiences on real machines

We have been running for several years now on various machines, although we did not do our largest runs until August 2009.

Lguest is now our VM of choice, running in 32-bit, even on 64-bit machines with hardware VM support. It has a small memory footprint and the kernel-based switching from the guest VM to the user-mode support code seems more efficient than the heavyweight context switch to KVM's QEMU support process.

This result has implications for many-task computing. If computing sites wish to run each task of a many-task run in its own VM, with high efficiency, it is worth looking at whether each of the apps can run in a 32-bit address space. If so, use of the VM support hardware may not be worthwhile. We are also looking at creating a lower-overhead support process for KVM guests than QEMU; we don't need most of what QEMU does, particularly the graphics support.

Migrating has no use to us in for Megatux, and we have doubts about its value in real HPC applications. We say this having run the BProc system for ten years: BProc supported very fast migration across the cluster, and we never found it useful in support of resilience and fault-tolerance. It is easy enough to move a process or a VM; what's hard is dealing with all the related external state, particularly network connection information in the switches, that has to be recreated. Infiniband and RDMA would be particularly problematic. Multiply the scale by 1000, as we are doing, and the problem looks even harder. Better to have a programming model which tolerates the disappearance of a VM and soldiers on than halt all 9,999,999 VMs while the missing one is resuscitated and all hosts and switches are updated.

6. Conclusions

We described Megatux, a set of tools which we use to rapidly deploy virtual machines. We are using Megatux on a range of systems, some six years old, others very new. Even on systems with hardware virtualization, we have achieved the best performance with Lguest in 32-bit mode with PAE enabled. KVM is in principle more efficient, but in practice, due to its QEMU support process, is far less efficient. Our tools are new and designed to be self-configuring, given that even the smallest configuration files on 1000-node systems balloon to unmanageable size on ten million node systems. We use a technique called computational configuration, in which parameters are set by an algorithm, rather than a configuration file entry. In the future system builders must structure their systems with computational configuration in mind; many of them are not doing so today.

Acknowledgments

Sandia National Labs paid for this via LDRD funding.

References

[1] R. Armstrong and J. Mayo. Emulytics: Large-scale emulation of botnets. Presented at SC 09 ACS exhibit, Nov. 2009.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945462. URL `http://www.cl.cam.ac.uk/research/srg/netos/papers/2003-xensosp`

[3] M. J. Brim. Tbon-fs: A new infrastructure for scalable tools and runtimes.

[4] A. E. Darling, L. Carey, and W. chun Feng. The design, implementation, and evaluation of mpiblast. In *In Proceedings of ClusterWorld 2003*, 2003.

[5] E. Hendriks and R. G. Minnich. How to build a fast and reliable 1024 node cluster with only one disk. *Journal of Supercomputing*, 36:171–181, 2006.

[6] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. Freiling. Measurements and mitigation of peer-to-peer-based botnets: A case study on storm worm. 2008. URL `http://www.usenix.org/event/leet08/tech/full_papers/holz/holz_`

[7] B. Labs. Introduction to the 9p protocol. *Plan 9 Programmer's Manual*, 3, 2000.

[8] R. Minnich, A. Mirtchovski, and L. Ionkov. Xcpu: a new, 9p-based, process management system for clusters and grids. In *Cluster 2006*, 2006.

[9] R. Russel. lguest implementing the little linux hypervisor. 2007.

[10] R. Russell. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1400097.1400108.

[11] M. J. Sottile and R. G. Minnich. Supermon: A high-speed cluster monitoring system. In *CLUSTER '02: Proceedings of the IEEE International Conference on Cluster Computing*, page 39, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1745-5.

[12] U. student. personal communication, 2010.