How To Build A Fast And Reliable 1024 Node Cluster With Only One Disk

Erik A. Hendriks
hendriks@lanl.gov

Ronald G. Minnich
rminnich@.lanl.gov

Advanced Computing Laboratory
Los Alamos National Laboratory*

Abstract

In the last year LANL has constructed a 1408-node AMD Opteron cluster, a 1024-node Intel P4 Xeon cluster, a 256-node AMD Opteron cluster and two 128-node Intel P4 Xeon clusters. Each of these clusters is controlled by one front-end node, and each *cluster* needs only one disk *in the front-end node* for production operations. In this paper we describe the software architecture that boots and manages these clusters. This software architecture represents a clean break from the way that clusters have been set up for the last 14 years. We show the ways that this architecture has been used to greatly improve the operation of the nodes, with particular emphasis on improvements in boot-time performance, scalability, and reliability.

1 Introduction

Traditionally, clusters have been built as collections of independent workstations. While these clusters are typically "racked and stacked" in one room, they are no different from the management perspective than a building full of machines. All the same headaches apply, from initializing the per-node disks, to ensuring that the disks stay up-to-date, to dealing with all the problems that occur when a disk gets out of sync or fails. Management systems such as Oscar [10] and Rocks [2] have been created to reduce the overhead of managing these groups of workstations. These systems succeed in reducing the cost related to managing a cluster of workstations but they do not solve the fundamental problem with clusters: there is software which needs to be maintained on every node in the system. Diskless NFS-rooted cluster systems such as C-Plant [8] mitigate this problem by consolidating the software maintenance for the nodes on a single or small number of servers.

*Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36. LANL LA-UR-03-9081

This helps with the disk management issues but adds another set of problems in keeping all the NFS roots synchronized.

Our goal is to reduce the software maintenance of a cluster to the point where it is no worse than a single multi-user system – with zero additional maintenance for the nodes. Additionally, in order for any solution to this problem to be relevant it will have to scale well to the large clusters being built today. We have set our scaling goals at about 1024 nodes. This is currently big enough to be relevant and also happens to be the size of our testbed.

Our approach to this problem involves building the simplest compute node possible — one with no software whatsoever beyond the system firmware. No software means that it will be possible to have diskless and therefore stateless nodes. This does not preclude having a disk — it means that no problem involving the disk will prevent the node from coming up.

BProc [3] is the basis that our cluster is built on. We will describe a light weight node design using BProc and then how to boot the node and get BProc running on it.

2 BProc

The BProc cluster management software has allowed us to build a compute node with no software installed on it. BProc is a Linux kernel modification which adds support for remote process control and process migration. On a cluster running BProc, there is a single front end machine and number of compute nodes. All processes are started on a front end machine and then migrated to a compute node where they run. When a process migrates, it remains visible in the process tree on the front end machine. Remote processes can be manipulated using standard UNIX interfaces (*e.g.* kill, wait).

BProc's process management and placement features are features typically associated with Single System Image (SSI) type systems such (*e.g.* OpenSSI, Mosix [1]). BProc provides these two features but that's where it ends. Unlike full blown SSI systems, the processes running on remote nodes see the node's local file system and the node's local network hardware. The environment provided to an application on a BProc cluster is very similar to that of a traditional pile of PCs type cluster. Each node has its own file system. Libraries such as MPI are used for interprocess communication.

The performance impact of providing other SSI features (*e.g.* single file system, single network, single memory space) is typically very high and severely limits the scalability of such systems. Scalability of these systems is usually measured in the 10s of nodes. Since BProc does not attempt to provide these features it can scale well beyond 1024 nodes with a single front end node.

On our cluster, BProc is the only method provided to start processes on remote nodes. Everything that gets run on the nodes (by users or administrators) is started using BProc process migration facilities. This includes applications like MPI jobs and system tools that an administrator might use like `mount`. Since BProc is the only method of interacting with the nodes, all the

infrastructure for supporting the usual interactive logins (authentication, daemons to accept logins and most of /bin) becomes unnecessary.

Although many users and some administrators believe they need them, interactive login services such as rsh and ssh are unnecessary. Traditional super computers don't provide these services. Users have simply become used to seeing them on clusters. All that is required is a robust mechanism to create and control remote processes. BProc provides these mechanisms. In fact, using BProc ends up being simpler in some cases since users no longer have to worry about paths and finding binaries on the remote nodes.

Since everything that the node runs comes from the front end, essentially nothing is required in the file system on the node itself. The only exception to this are a few shared libraries. This is an artifact of that the fact that dynamic linker gets run on the nodes and needs to be able to find its libraries. The bulk of these libraries is small enough to be placed on the node each time the node boots. Requiring no software installation allows for an extremely simple compute node.

BProc consists of some kernel modifications, a kernel module and a user space daemon that must be run to connect the node to the front end node. BProc does not provide a mechanism to get itself loaded. Therefore we must provide a scalable and reliable mechanism to get the nodes booted, the network hardware configured and the BProc daemon started.

3 Booting the Nodes

To get BProc started on the compute nodes we use a fairly straight forward network boot scheme with a significant twist – a Linux kernel placed in the system firmware will be our boot loader. The system firmware is a solid-state device, orders of magnitude more reliable than a hard drive.

Booting a cluster node starts with the system firmware. The system firmware will have to be capable of diskless booting. Commercial BIOSes provide a number of options for diskless boot but none of them are adequate for our purposes. We will describe the problems commercial BIOSes have in a cluster setting and then the solution — replacing it with LinuxBIOS and how we implement a diskless boot on top of it.

3.1 Firmware

Commercial BIOSes tend to be very configurable, and the configuration parameters are completely undocumented. The configuration is stored in CMOS. The BIOS writers have for years assumed that any modifications to the CMOS settings will be done interactively, via keyboard, monitor, and (they think this is an improvement) a mouse. If anything goes wrong with the BIOS settings, the only option is to visit every node with keyboard, monitor, and mouse, and change the settings manually. This gets tiring by the fiftieth node. By the time one gets to the 1024th node, a different career path starts to look very attractive.

Even if the CMOS parameters on a given node are correct, we have found that they degrade over time. Some CMOS errors will configure a machine so that it will not boot. For example, on

one set of Compaq DL360 servers we had, the systems would tend to forget that we had told them they had no keyboard. Hence, about one time in ten, the nodes would come up with the message *No keyboard - hit F1 to continue*. The only way to repair the node at this point was to attach a keyboard and monitor to the node and reset the "ignore keyboard error" flag.

Finally, network boot support is not available for all the different types of network hardware that Linux supports, such as Myrinet, Quadrics, SCI, and InfiniBand. The network boot that is supported in the BIOS is in general useless for clusters. In our experience PXE bioses are become unreliable when even small numbers (10s) of nodes are involved.

The BIOS update situation for standard BIOSes is terrible. Many of them require a DOS floppy. In many cases the update program requires interaction, which again requires keyboard, monitor, and mouse. There is no path to updating the BIOS from Linux.

For these and other reasons, we started at the bottom by replacing the commercial BIOS with LinuxBIOS [7]. LinuxBIOS is an open source replacement BIOS which replaces the vendor supplied BIOS in *flash RAM*. It is a small hardware setup program which configures the CPU, memory, I/O hardware and then loads some payload. The payload can be almost anything — a boot loader such as Etherboot [11] or FILO [9] or even a Linux kernel.

All LinuxBIOS upgrades can be done from Linux. All CMOS settings are documented and can also be set from Linux. In contrast to commercial BIOSes, there are no CMOS settings that will cause LinuxBIOS to fail to load its payload.

LinuxBIOS provides us with the absolute control of the node that we require. It supports failsafe boot and a fall-back BIOS image. It configures just as much of the hardware as Linux needs configured, but no more. Finally, it allows us to load many different types of images, including a full Linux kernel and initial RAM disk, into Flash.

3.2 Generic Network Boot With Linux

The payload in our case is a Linux kernel and initial ramdisk (initrd) image. The firmware kernel and initrd will act as a boot loader to load another operational kernel from a boot server. The operational kernel is not placed in flash because that would require a flash update every time the kernel was upgraded — a risky proposition at best.

Instead we place a stripped down Linux kernel in flash which contains just enough device support to download another kernel and ramdisk image from a boot server. Once downloaded, it replaces the running kernel with the new kernel using *Two Kernel Monte* [6]. This is essentially a network boot loader implemented with Linux.

Using Linux here offers a number of significant advantages over using other network boot systems such as PXE [5] and Etherboot [11]. If Linux is the boot loader then the set of network hardware supported by the boot loader is exactly the same as your final operational kernel. This includes exotic networks such as Myrinet and Quadrics. This has allowed us to build diskless clusters with only two wires to each node - the high performance network cable and power. Our 1024 node testbed Pink was constructed this way — Myrinet is the only network.

Activity	*Time (secs)*	*Total Time (secs)*
Power-on to memory ready, ECC scrubbing done, etc. Highly variable due to power supply timing. Also depends on how much memory in the machine. Measured for 1 Gbyte.	2.23	2.23
CPU and other hardware initialization.	0.35	2.58
Search FLASH memory for ELF Image.	0.13	2.71
Linux Kernel Loaded from Flash.	4.15	5.85
Linux Kernel initializes hardware, loads Myrinet driver and stored route from CMOS, and sends first RARP.	1.3	6.15
Full Myrinet routes loaded from front end.	1.1	7.25
New kernel and initrd downloaded. Includes retransmission time of 5s.	8	15.25
Phase 2 kernel booted.	1.7	16.95
Phase 2 initializes hardware, loads Myrinet driver and stored route from CMOS, sends first RARP.	11.3	28.25
Phase 2 loads Myrinet routes.	1.1	28.35
Front-end runs node setup program on node, adds plugins and does housekeeping chores.	5	33.35
Node completely up.		33.35

Figure 1: Single node boot timing for a node in Pink. This table shows the time required for each step in booting a single node in Pink. The first step starts at power-on and it continues through to the end of node setup. Once the final step is completed the node is ready for a user to use. This system boots off its Myrinet network. We have replaced Myricom's mapper software with our own mapper *gm_route* [4]. Our mapper allows us to store the route to the front end node in CMOS. This allows the node to get a network map and generate routes without searching the network.

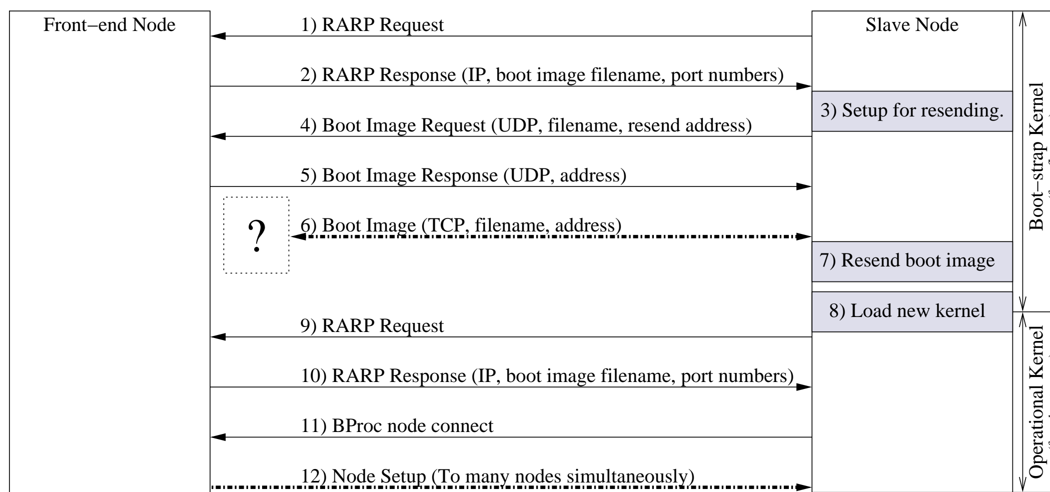

Figure 2: The boot process. The first eight steps are carried out by the phase 1 (bootstrap) image loaded from the firmware. The remaining steps are carried out by the phase 2 (operational) image.

Another nice side effect of using LinuxBIOS is that it is much faster than most commercial BIOSes. Figure 1 shows timing for every step involved in booting a node in 'Pink'. The Linux kernel in firmware is loaded and running in just under 6 seconds after power-on.

Linux also provides a very featureful environment in which to implement the client software for downloading the new kernel. For example, our boot image downloader involves TCP. Using Linux's implementation meant that we didn't have to re-implement reliable streaming of data with flow control.

The boot strap program is a normal user space program that runs under Linux. Figure 2 shows the sequence of events involved in bringing up a slave node. Since it's a user space program, the boot strap is both easy to debug and can take advantage of all the features that Linux has to offer.

First the network hardware is configured using RARP (Figure 2 steps 1,2). RARP was chosen because it is simpler than DHCP and we have no need for DHCP's more advanced features. The RARP response tells the compute node where the front end node is in the network. The client then downloads the boot image from the front end. The boot image is a kernel plus an initial ramdisk. The boot image usually starts at approximately 1MB and grows depending on how many drivers are included. Some drivers (*e.g.* Myrinet) can be quite large. On our systems, the file normally weighs in at about 2MB.

In order to reduce the load of sending boot images on the server, the client nodes are used as servers after they get a copy of the boot image. Reusing this clients in this way ends up building an ad-hoc tree structure for data distribution.

Before a client requests a copy of the boot image from the boot server, the client picks an

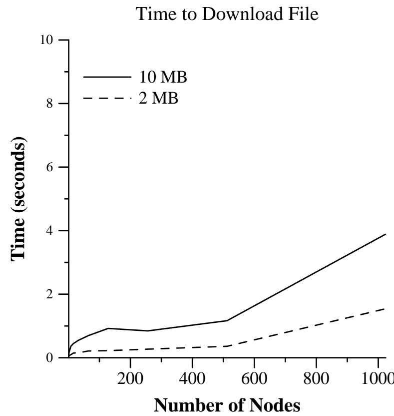

Figure 3: Time to download a file with the boot image file distribution mechanism. Note that these times do not include the retransmission time that's built into the protocol. This test was performed on a 1024 node Myrinet network.

address and port number that it will resend the image on (Figure 2 step 3). This address and port number is sent to the boot server along with the request for the boot image (Figure 2 step 4). The server makes note of the time that the client made its request and adds it to a list of possible senders for future requests. The server then responds to the client with an address to download the boot image from (Figure 2 step 5). The address in the response could be the address of the boot server itself or one of the previous clients. The server also includes other instructions for the client such as how long to make itself available for resending the image. The client then connects to the address given in the server's response (Figure 2 step 6) and downloads its copy of the boot image. Then waits for a short period of time (5s) in case it gets any resend requests (Figure 2 step 7).

The initial request and response from the server node are implemented with UDP. The client connects to a server to download the boot image with TCP. If any of these steps fails, the client goes back to asking the server where to download the boot image from. The ad-hoc nature of this tree structure allows it to adapt easily in the face of failures.

Figure 3 shows performance results for boot image download on the 1024 port Myrinet network. Note that the times in Figure 3 do not include the retransmission time that's built into the protocol. The time send a typical 2MB boot image to all 1023 nodes is 1.5 seconds.

Comparison with other download schemes

This mechanism for distributing data replaced a previous system which used multicast or broadcast to distribute the boot images. Multicast is a popular mechanism for mass data distribution but it depends on support in the network hardware to perform well. Ethernet does broadcast well but

many switches can't handle multicast well or they just treat it as broadcast traffic. When using exotic network hardware (*e.g.* Myrinet) even good broadcast support is not certain.

The network that forced us to abandon the previous system and implement a new one was the 1024 node Myrinet network in 'Pink'. The Myrinet is the only network in Pink so it must be used for all boot-up and management functions.

Myrinet has no support for multicast or broadcast in hardware. In GM 1.6.3 (the newest driver version available at the time) broadcast was implemented as a series of sends to every node in the network. This reduced the effective network bandwidth to 2Mbps or 1/1024 of what it normally is. Booting the entire system at once was more or less impossible since the front end node became starved for bandwidth.

The new boot image download scheme fixed this problem since it uses *only* point-to-point communication. It follows that the new scheme should work well on any network which is good at supporting a lot of simultaneous point to point links. In other words, any switched network.

Somewhat unexpectedly, the new download scheme performed better on our Ethernet connected clusters as well. This appears to be due to the fact that Linux's TCP implementation has better flow control and adapts to the available bandwidth more effectively than our own multicast download implementation. Surely, this is a short coming of our own multicast system but it illustrates the benefit we're getting by using the existing code base in Linux.

3.3 The Operational Kernel

Once the boot image is downloaded, it is loaded into memory using Two Kernel Monte. Two Kernel Monte is a Linux based boot loader. It overwrites the running Linux kernel with a new one and transfers control to the new one (Figure 2 step 8). The new kernel from the boot image is the operational kernel that we eventually want to be running.

The phase 2 image starts out similarly to the second one. It first configures the networking hardware using RARP (Figure 2 steps 9,10). Instead of downloading a boot image, it starts the BProc slave daemon. The BProc slave daemon connects to the master node (Figure 2 step 11). Once this is done, the master can send processes to the slave. From the slave's point of view, setup is finished at this point. Further setup is handled from the master node. The only way to interact with the node at this point is through BProc and that means by migrating a process to the node.

The boot up program included in the initrd cleans up the node as much as possible before starting the BProc slave daemon. When the boot program starts, the root file system is the initrd image which is a `romfs` file system that contains the boot program, a few device nodes and drivers. The boot up program uses `pivot_root` to make a `tmpfs` file system the root file system. After driver loading is complete, the `romfs` file system is unmounted and the memory is freed. When the BProc slave daemon is started, the root file system on the slave node is an empty `tmpfs` file system. The only thing running on the node is the BProc slave daemon. The boot up program becomes `init` for the node.

When slave nodes connect to the front end, the front end node automatically starts a process

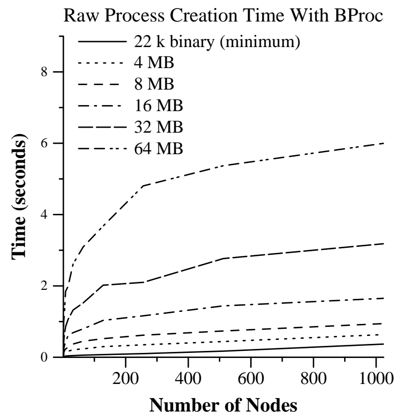

Figure 4: Process creation time with BProc. These tests were performed on a 1024 node Myrinet connected cluster. A single process is replicated on many nodes – one process per node.

to setup the new slave node. It's the responsibility of this program to setup the slave node and make it usable to users (Figure 2 step 12). Typically, a few file systems need to be mounted on the node (*e.g.* `/proc`) and a few shared libraries need to be copied to the node.

Process migration is the only way to interact with the slave node. The node setup program loads everything that it will need to setup a node (configuration information, libraries, etc.) into its memory space on the front end. Then it migrates to the node and dumps out the contents of its memory space onto the node. Typically the size of the process image ends up being approximately 5-100MB depending what needs to be on the node. This data is typically stored in the `tmpfs` root file system. On our clusters, the nodes typically have 2GB of memory or more so this represents about 3% of system memory. The actual cost is less since these files would be memory mapped and paged in by processes running on the nodes anyway.

Copying this large process to every node would be very expensive. BProc offers a facility to efficiently make many copies of the same process image on many nodes. Internally it is very similar to the ad-hoc tree system that is used for distributing boot images. The only difference is that each process is available to resend the process image until all processes are created instead of some fixed time. Figure 4 shows process creation performance result for BProc. It shows the time required to make copies of a single process to many nodes in the system. Time required is $O(logn)$ where n is the number of processes. It takes 0.36 seconds to create 1023 22KB processes on 1023 slave nodes. 22KB is the smallest process image we could create. It takes just under 6 seconds to do the same for a 64MB process. Since there is a single front which is keeping track of all the remote processes we expect that it will be come the bottleneck in the end.

Due to the fact that the node setup program takes the bulk of the data with it as part of the process migration, it is possible for the node setup program to take advantage of BProc's tree

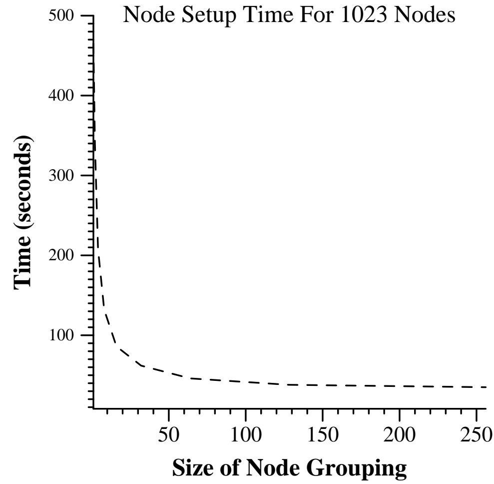

Figure 5: Time to perform node configuration on all 1023 nodes of Pink. The node configuration system setups up the nodes in groups. This graph shows how the time to configure all 1023 nodes in different size groupings.

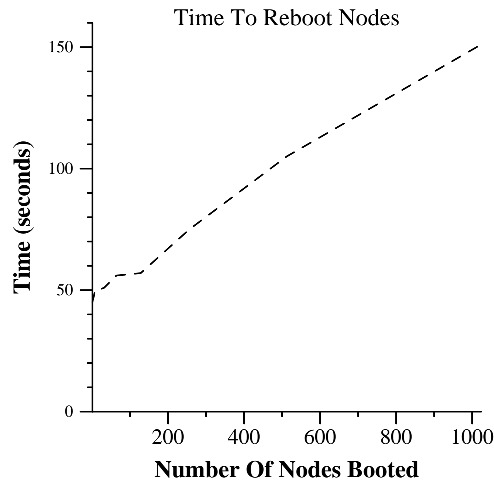

Figure 6: Time to reboot nodes on Pink. The time measured is the time from power-on on the nodes to the time when the nodes were ready to be used by users.

spawn mechanism. The only additional requirement is that node setup is performed in groups. A daemon on the front end node collects requests to setup nodes. Once there is a large enough group of nodes waiting to be setup or enough time has passed, the daemon starts node setup for all the nodes at once. The delay before going on with a smaller group is currently 1.5 seconds but this is easily tunable. Figure 5 shows the effect that grouping the nodes has on node setup performance. The graph shows the time required to setup all 1023 slave nodes with different sized node groupings. The size of the node setup process in this case is approximately 55MB. The bigger the node grouping the better the performance. We did not test beyond a grouping of 256 nodes because the node setup starts to run out of file descriptors at that point. It takes 35 seconds to setup all 1023 slaves with a node grouping of 256 nodes.

After node setup completes we have a functioning cluster node. It is relevant to note that there is no "installation" procedure or "upgrade" procedure for the nodes. Both are simply a reboot. If a node gets messed up for any reason a reboot will fix it. Most of the time simply re-running the node setup program without a reboot will fix it as well.

Figure 1 shows the timing for a single node in Pink to boot from power-on to when the node is ready for a user to use. There are a few extra steps due to Myrinet but the node is still up and ready to use in about 33 seconds. Figure 6 shows the overall time required to reboot portions of Pink. The time measured is the time from power-on on the nodes to the time when the nodes were ready to be used by users. The entire cluster (not including the front end node) can be rebooted and ready to use in 151 seconds.

4 Using The System

From a user's point of view, running applications is a lot like using a single system. Everything is started on the front end. Programs such as mpirun handle the details of placing processes on nodes. Users can check the status of running jobs using `ps` and kill them with Ctrl-C or `kill` as if they were local. A user can even attach a debugger like `gdb` to a remote process as if it were local. Since all jobs are visible on the front end users are also much less likely to have runaway jobs.

The biggest hurdle that our users have encountered is adapting existing scripts to the new system. Scripts that use things like `rsh` to interact with the nodes directly will not work without modification. However, the changes required are usually not drastic. Some times it's possible to simply substitute BProc's remote execution program (`bpsh`) in `rsh`'s place. User education is usually a bigger problem than making old scripts do what they need to do in the new environment.

Developers and administrators encounter the bulk of the problems adapting existing tools to the new environment. Things that qualify as "tools" tend to be more complex and require more work. For example, we were forced to modify the startup code for MPICH (p4 and GM) to work in this environment. This was not a trivial job but taking advantage of BProc's fast process creation capabilities led to a massive improvement in startup performance. As more of these types of systems get deployed, there should be less and less of this type of work to do.

Another issue for administators is file system configuration. BProc does not address the problem of a global file system. This is partly intentional since it's a hard problem to solve and there are already several competeing network file systems. However, since each node has it's own file system, anything can be mounted on the nodes.

Our systems mount some kind of network file system and may also have local disk based scratch space mounted. This allows BProc system to use any of the competing network file systems. NFS, Panasas and Lustre have all been used successfully on BProc systems.

5 The Real World

This software has had enough time to see a fair bit of real world usage. It has proven to be quite practical on a large scale. It runs the largest AMD Opteron cluster in the world, as well as some of the larger clusters at LANL. A partial list of the clusters includes:

Lightning
: a 1408-node, dual processor AMD Opteron / Myrinet cluster

Blue Steel
: a 256-node, dual processor AMD Opteron / InfiniBand cluster

Pink
: a 1024-node, dual processor Intel P4 Xeon / Myrinet cluster

Grendels
: a 128-node dual processor Intel P4 Xeon / Myrinet cluster

Ed
: a 128-node, mixed Alpha / Myrinet cluster

Pinkish
: a 128-node, dual processor Intel P4 Xeon / Myrinet cluster

Lightning, Pink and Grendels are production systems. The others are research systems. Running a cluster in this way makes a dramatic difference in the effort of maintaining and using a cluster. It takes about one tenth the manpower to run these new clusters as it takes to run our older Tru64-based clusters.

6 Conclusion

Obviously, it has proven to be practical for use at LANL. Clusters built this way are usually up and running within a few days of arrival, which is far faster and easier than previous clusters. The

clusters perform well — they are now #11, #211, #218, #311 on the 23nd Top 500 list as of June 2004. #211 (Catalyst) is at Sandia National Laboratory in Livermore, CA.

Users are very happy with this model. The systems stay up and perform well. Users have found that scheduling, job startup and job control on these clusters are far faster than what they are used to.

The system administrators have found that these clusters let them sleep at night. No one will ever have to manually fix every node because /etc/passwd just disappeared, or the C library got corrupted, or /sbin/init is gone. Any problem that a node has can be fixed with a reboot, or in the worst case, a power cycle. It's trivial and safe to do things like swap kernels.

They are also very practical for system providers. Vendors have told us that this type of cluster is far easier to build and install than any other type of cluster. The cost to the vendor of delivering this type of cluster is much lower than traditional clusters. The cost to the vendor of making this type of system work is also much lower. Vendors have an economic incentive to build clusters using this software technology.

7 Availability

All the software described in this paper is freely available under the terms of the GPL. This system is the result of three distinct software packages. LinuxBIOS (the open firmware) is available from `http://www.linuxbios.org`. BProc (the process creation and management system) is available from `http://sourceforge.net/projects/bproc`. The boot up software described is a modified version of the Beoboot package which is also available on the BProc site. Two Kernel Monte is a part of the Beoboot package.

Binary packages (as well as source) of all this software are also available from `http://www.clustermatic.org` as part of the *Clustermatic* software package.

References

[1] Amnon Barak, Oren La'adan, and Amnon Shiloh. Scalable cluster computing with MOSIX for Linux. In *Proceedings of the Linux Expo '99*, pages 95–100, Raleigh, NC, May 1999.

[2] Greg Bruno and Philip M. Papadopoulos. NPACI Rocks: Tools and Techniques for Easily Deploying Manageable Linux Clusters. In *Proceedings of Cluster 2001*, Anaheim, CA, October 2001.

[3] Erik A. Hendriks. BProc: The Beowulf distributed process space. *16th Annual ACM International Conference on Supercomputing*, June 2002.

[4] Erik A. Hendriks. Fast mapping on myrinet networks. In *Proceedings of 7th International Conference on High Performance Computing and Grid in Asia Pacific Region*, July 2004.

[5] Intel Corporation. Preboot execution environment (PXE) specification. 2002.

[6] Ron Minnich. Give your bootstrap the boot: Using the operating system to boot the operating system. In *Proceedings of Cluster 2004*, San Diego, CA, October 2004.

[7] Ron Minnich, James Hendricks, and Dale Webster. The Linux BIOS. In *Proceedings of the Fourth Annual Linux Showcase and Conference*, Atlanta, GA, October 2000.

[8] Rolf Riesen, Ron Brightwell, Lee Ann Fisk, Tramm Hudson, Jim Otto, and Arthur B. Maccabe. Cplant. In *Proceedings of the Second Extreme Linux Workshop*, June 1999.

[9] SONE Takeshi. Filo readme. Technical report, http://te.to/ ts1/filo/.

[10] The Open Cluster Group. OSCAR: A packaged cluster software stack for high performance computing. January 2001.

[11] Ken Yap and Markus Gutschke. Etherboot User Manual. July 2002.