A Minimal Linux Environment for High Performance Computing Systems

James H. Laros III*[1], Cynthia A. Segura[2], and Nathan Dauchy[2]

March 16, 2006

Abstract

This paper describes the use of standard Linux®[3] and Open Source software to produce an environment to support parallel scientific applications on High Performance Computers (HPC). The goals of this approach are to maximize the HPC resources delivered to the application, to improve system stability and predictability, and to reduce software management burdens. The simplicity of this approach provides an additional benefit. The paper presents the reader with background, motivations, and a discussion of advantages and drawbacks of the light-os.

Keywords: Linux, Open Source Software, Clusters, High Performance Computer, Light Weight Kernel

1 Introduction

This paper describes the use of Linux and Open Source software to produce an environment designed to be run on nodes (which we call compute nodes) that support High Performance Computing (HPC) applications[†]. This approach, which we call light-os, reduces system overhead to a bare minimum so that as many resources as possible are devoted to the scientific applications that run on the system. Each compute node will host only the light-os, a run-time component (such as described in Section 5) and the HPC application. For a more detailed list of requirements for the light-os component see Section 2 (Related Work).

The light-os draws upon research at Sandia National Laboratories[1] in the areas of Light Weight Kernels (LWK)[4] and disk-less Linux clusters. Sandia has a long history in research and use of Light Weight Kernels on large HPC systems with up to 10K nodes, and has also implemented disk-less Linux clusters with up to 2K nodes[5][6][7][8]. This work targets a gap that we feel exists between

*Corresponding Author James H. Laros III, jhlaros@sandia.gov

†We sometimes use the more general term HPC rather than Linux cluster, because many of the concepts discussed here are derived from and/or are applicable to systems that are not generally considered Linux clusters.

the LWK environment on capability systems and the disk-less Linux Operating System (OS) implementation on our capacity systems. While our traditional disk-less Linux implementations were light we feel that by taking a more focused approach we can eliminate many of the requirements imposed on the operating system and thereby allow the light-os environment to more closely emulate a custom LWK environment. By leveraging Linux in this way we hope to achieve many of the benefits of a LWK with far less effort[‡].

The original motivation for this work emerged from research funded in 2003 by the Computer Science Research Foundation (CSRF)[9] at Sandia Labs. The initial effort involved a survey of the current state of embedded systems, software and hardware, to determine the practicality of designing a Reliability Availability and Serviceability (RAS) system based on embedded technology. The light-os work was developed to mimic loading a LWK to a large number of nodes from a single embedded device. (Porting a LWK to the test architecture was far beyond the scope of the project.) By leveraging some of the new (at the time) additions to the Linux kernel, like tmpfs[§], implementing what eventually became the light-os was surprisingly simple.

After serving the original purpose, it quickly became evident that this approach could be useful for HPC systems in general. In Section 2 (Related Work) we detail the criteria for the light-os and provide a survey of related efforts. In Section 3 (Argument for this Approach), we describe the logic and reasoning behind the light approach. Section 4 provides an overview of the light-os environment. While not directly part of the work described in this paper, runtime, and systems management are integral parts of any HPC system. A few examples and their relationship with light-os can be found in Section 5 (Runtime) and Section 6 (Systems Management). Clearly, there are advantages and disadvantages to this approach. We discuss some of the drawbacks and considerations where appropriate. Some thoughts on future work can be found in Section 7 (Future Work). In Appendix A, we provide an example implementation of the light-os using generic and popular runtime components.

2 Related Work

In our short survey we found some interesting projects that relate, at least somewhat, to the work described in this paper. The following are the requirements we set out to meet with this effort which will aid in the discussion of these related efforts.

- No local disk - The compute nodes will not require or, even if present, use a local disk for the root or any other file-system including swap. Swap space will not be available to the application.

[‡]It is important to note that we do not expect even a stripped down Linux environment to be as scalable or efficient as a LWK implementation on a well balanced platform.

[§]tmpfs is a file system, which according to the 2.6 kernel documentation, "puts everything into the kernel internal caches and grows and shrinks to accommodate the files it contains."

- No root remote file-system - The compute nodes will not require a remotely mounted root file-system for the purposes of system operation. In particular, no NFS-root file-system will be required. (This does not include access to a file-system for application input and output.)
- No dependency on a Linux Distribution - The light-os environment will not have a dependency on any specific Linux Distribution. Typically only Linux kernel source will be required.
- No kernel source modification - The light-os environment will not require any code modifications to the Linux kernel source. Tuning the kernel by existing methods is permissible.
- No Linux system daemons - The goal is to eliminate all non-application or runtime processes on the compute nodes.
- No static memory allocation for system use - No static memory will be allocated such as static ram-disk space for root file-system or run-time libraries.

The Warewulf project[10] stood out among other disk-less cluster efforts that we surveyed. In their online documentation we found many important guiding principles that we share. The authors seem to have come to many of the same conclusions as we have about how to best build a disk-less cluster[6]. While Warewulf is an interesting project, it does not meet all of our requirements; notably, Warewulf uses both a static ram-disk and an NFS-mounted root file-system. The documentation also seems to allude to a Linux Distribution dependency which they seem to be working to remove. While Warewulf is not a good fit to satisfy our goals we are interested in testing it locally for other purposes.

The Clustermatic[11] efforts, more specifically BProc, at Los Alamos National Labs[12] (LANL) is another effort that has some similar goals. BProc has been around for a long while and is probably best contrasted with the Cplant[5] work done at Sandia Labs. In comparison with the light-os work BProc shares the desire of minimizing the system environment that runs on the compute nodes. BProc requires kernel modifications and also uses a static ram-disk for shared libraries. BProc has proven to be a very successful effort at LANL but is a much more broad effort that does not fit well with the other components such as the run-time system, used in this effort.

While there are other disk-less cluster efforts or more aggressive full system solutions like Clustermatic, we did not find any efforts targeted to the area we are addressing, or components of larger projects that we could leverage. The light-os work is attempting to bridge a gap between clusters and more integrated HPC systems.

3 Argument for this Approach

The light-os philosophy assumes that, on systems designed to run scientific applications, as many computing resources as possible should be dedicated to the application running on the system. In turn, non-application software should consume as few system resources as possible, and should be limited to the software that is essential to support the application. By eliminating unnecessary overhead, the light-os makes more resources available to the application, and reduces the complexity and overhead of the system. A less complex environment should in theory require less maintenance and support, and could exhibit such advantages as increasing the Mean Time Between Failure (MTBF), decreasing application wall clock run times, reducing administrative requirements, and shortening boot times.

To illustrate, let us first compare the light-os approach to a standard Linux implementation. Both the light-os and the standard Linux environment depend on a Linux kernel, built from standard kernel source. In this area, the light-os displays an advantage in that it requires less hardware and software service support, which in turn can reduce the footprint[¶] of the running kernel. Since the kernel consumes less memory, more is available for the application. Additionally by reducing the complexity of the kernel, we expect an increase in the stability of the system. While MTBF is a metric that is commonly discussed in relation to hardware, it can logically be applied to software. As the number of hardware components increase, the MTBF of the overall system generally decreases[‖]. Similarly, as the number of software components increase, the more likely there is to be an error, thereby decreasing the MTBF of the system. By omitting unnecessary components, we reduce the potential for errors[**].

In a standard Linux environment there are many system services and daemons that take system resources and require administrative attention. The light-os environment runs few additional services and/or daemons, returning resources to the application and reducing complexity and administrative requirements. The additional system services present in a standard Linux environment impose an additional impact. HPC systems generally run large-scale parallel applications that often have barrier points that all nodes participating in the application must reach before progress beyond the barrier can be made. When nodes are interrupted by system services, progress among the nodes used in the application becomes unpredictable and it is possible, if not common, for many nodes to sit idle waiting for a single node to catch up[13]. By minimizing the number of processes on a node, the light-os approach reduces the frequency that the user application is interrupted. This may result in shorter wall clock run times, which increase overall system resource availability, and, more importantly to users, more predictable run times for their applications. The greater the complexity of the system, the greater the chances of failure. By reducing the processes running on the nodes, we likely increase the stability of the system.

A light-os implementation requires no Linux distribution. In contrast on a disk-full[††] cluster, some distribution mechanism must be employed to copy the

[¶]Footprint in this context is the amount of memory used.

[‖]For hardware components with less than an infinite reliability.

[**]By reducing components we mean both reducing the number of processes or daemons running and reducing components compiled into the kernel.

[††]The term disk-full is used to describe nodes (computers) that require disks to boot and

distribution to the disk-full nodes. The distribution mechanism requires some support, if not development cost. It is our opinion that, especially for HPC systems, disk-less nodes are superior for many reasons outlined in *Implementing Scalable Disk-less Clusters Using the Network File System*[6]. The light-os approach extends the benefits discussed in this paper by minimizing server involvement and removing the need to maintain a Linux distribution. In the case of disk-less clusters that use NFS as their root file-system, the node continues to be dependent on the server. By contrast, the light-os node becomes independent from the server node once it receives its environment.

For large HPC systems, boot time can become a real factor in system availability, especially if the MTBF is small. The light-os boot process is simple and fast, constrained primarily by the time it takes to retrieve the light-os package from the server. The node does not have to perform many aspects of the boot sequence, such as mounting additional file-systems, file-system checks, or initializing a large number of services. This process enables each node to be largely autonomous, and allows the boot process to be highly parallel throttled only by the ability of the server to deliver the light-os package[‡‡].

There are many ways to implement a light environment. One approach would be to use a LWK in conjunction with the same runtime system used in the light-os environment. If we contrasted our proposed minimal Linux OS with a LWK specifically designed for the target platform the LWK would likely prove to be superior in all cases except the effort to produce and maintain the LWK itself. At some scale this cost is certainly warranted. It is our contention that the light-os environment can provide value in the gap between traditional Linux cluster implementations and very high end custom platforms that warrant the additional expense of LWK development.

4 Light-os: In Brief

The light-os is simply a minimal Linux kernel running on a compute node. By minimal we mean only what is required to serve the runtime component (described in Section 5) and the HPC application. We start by building a Linux kernel from standard source code. There are no restrictions imposed on where the source comes from other than it must support the hardware that you will be running on. We have used source from kernel.org[14] and various other Linux distributions. During the configuration step of building the kernel we are careful to include only the components that are necessary to support the hardware that is present on the compute node that will host the light-os. Additionally we typically choose to build all capabilities into the kernel, which saves us from having to load them as modules later in the initialization process. For example, during the configuration process we would select the specific ethernet driver for the node hardware to be built into the kernel. Other than building the kernel for

operate as part of a system.

[‡‡]The use of tftp while not the only way to deliver the light-os package has been tested on a ratio of up to 256 to 1.

the specific hardware that will host the light-os, we need to choose appropriate options to be able to use an initial ram disk (initrd) and tmpfs. (Refer to Appendix A.2 for specific configuration options). The primary purpose of the initrd is to act as a container that will be used to ship everything necessary for initialize to the node. The linuxrc program (in our case typically a script) will orchestrate the initialization. This initialization will differ dependent on factors like what hardware is present or what run-time is being used but will follow the same basic sequence.

In the following sequence we assume the kernel that we built with the process described previously is delivered to the node along with the initrd. The node loads and initializes the kernel, notices that there is an initrd, mounts it and executes the linuxrc script. At this point we take control of the initialization of the node. Our first step is to establish what will be our root file-system, located in tmpfs, with the software and utilities that are contained in the initrd. We create only the devices and directories that will be absolutely necessary to support the runtime and application, and copy only the software that will be needed on the node for the long term. (We optionally load BusyBox[15] into the tmpfs root file-system since the trade-off of functionality versus space is acceptable to us.) Once constructed we pivot root into our new root filesystem. At this point we can load any kernel modules that were not able to be built into the kernel; for example, modules to support high speed network for the system. It is important to note that these kernel modules are still located in the initrd filesystem.

At this point we can accomplish any additional setup required. Keep in mind that any scripts or temporary files remain in the initrd filesystem. When setup is complete, we unmount the initrd filesystem to free up as much memory as possible. This allows us the luxury to be less careful about what we put in the initrd since it will not remain past the system initialization step. The final step is to execute chroot. In our case we pass our runtime program as a parameter to chroot to be executed. The runtime program becomes the only process running on the system when initialization is completed.

These are the basic steps taken during the initialization of a node running the light-os in conjunction with a runtime system designed for HPC. The design of this runtime system allows us to strip down the Linux OS to this minimal state. Taking this approach does present challenges. Some of the trade-offs to be considered are discussed in Section 5 Runtime, and in Section 6 Systems Management. In Appendix A we describe a more generic approach, using additional Open Source packages, that we hope will provide an easy way to experiment with these concepts.

5 Runtime

The light-os does not require this specific runtime system, but a runtime system with these characteristics that is specifically designed for HPC systems is the optimal compliment for the light-os*. A discussion of all of the characteristics of the runtime system is beyond the scope of this paper, but we will provide enough detail to familiarize the reader with the basic functionality provided.

Sandia Labs has produced a long lineage of HPC systems with scalable runtime components. Many of these systems have used a LWK. Cplant[5], however, implemented the same runtime components on a commodity cluster system using a typical Linux OS. The runtime components that we will discuss perform the same basic function whether implemented for use with a LWK or a commodity Linux kernel. The components of the runtime system important for this discussion are named pct and yod. Pct is the runtime component that is executed during system initialization (Section 4) on each compute node. The pct process remains persistent for the life of the node. Yod is the command used to launch the HPC application. In the most basic usage of yod, an application is executed on the system by using the yod command combined with a flag that specifies the list of nodes the HPC application should be launched on. Yod communicates with the pct processes on all of the nodes on the list and distributes the application to each of the nodes. In our implementation the application executable is effectively copied, although in a scalable fanout fashion, to each nodes filesystem. Recall from the discussion in Section 4 that each node's filesystem is a memory resident filesystem. Once the fanout process is complete the pct starts the application. After the application is started pct basically gets out of the way to allow the application the maximum amount of node resources possible.

One trade-off to this approach is that statically compiled executables are required. The light-os environment, similar to the LWK environment, does not make shared libraries available to applications. Another trade-off is that if the above method is used to distribute the executable to each node, both the running executable and the executable file take up memory resources. The LWK uses a more sophisticated approach which does not cause this effective duplication. An alternative approach is to make the application executable available to each node via a parallel filesystem, such as Lustre[20, 16], the Parallel Virtual File System[17], Panasas[18] or IBRIX[19]. Although the use of a scalable parallel filesystem is essential for input and output this method of launching an executable also presents trade-offs and is beyond the scope of this paper. Another point worth mentioning is that while the runtime system we have discussed is freely available it is not as ubiquitous as the MPI runtime system.

6 Systems Management

Because the light-os does not have any daemons or services, it requires an "out-of-band" monitoring solution; namely, the management and monitoring of the node should be done from some point external to the node with no host processor involvement. An out-of-band monitoring solution ensures that a user's

*As demonstrated in Appendix A.6, a light-os implementation can accommodate a typical Linux MPI runtime environment.

application will not be interrupted by extraneous processes.

Ideally, the light-os would be used in conjunction with a Reliability, Availability, and Serviceability (RAS) system like those found on Red Storm[21] or Blue Gene[22]. These systems provide specialized out-of-band monitoring to to support custom hardware. Currently there is no RAS solution for the light-os, although research in this area is well underway[23]. Many vendors are introducing primarily out-of-band alternatives; for example Hewlett-Packard's Integrated Lights Out[24], Sun's Advanced Lights Out Manager[25], and Dell's Dell Remote Access Card[26]. These solutions provide basic capabilities for system control, such as the ability to power cycle a node. Unfortunately, these solutions are typically proprietary and are only useful on the specific system for which they were designed. In contrast, the Intelligent Platform Management Interface (IPMI)[27] provides a feasible Open Source out-of-band monitoring solution to manage a node. IPMI also provides out-of-band access to detailed information about the general health of the system, including things such as temperature and fan speeds.

The light-os approach reduces or eliminates non-application processes, which can in turn impact the capability to manage or monitor a system. Although, the light-os requires an out-of-band monitor solution, it also reduces the need for monitoring; for example, the light-os nodes do not need to track disk capacity and health or monitor running processes (the runtime system monitors the activity of the HPC application). Similarly, since there are fewer software components, there is less opportunity for problems. (For a discussion of the advantages of the light-os see Section 3).

7 Future work

The most important work yet to be accomplished is to test this concept on as large a system as possible for long durations. While some of the thoughts put forward in this paper might seem logical, they lack the empirical evidence that can only be provided by testing. During initial development, these concepts were tested with the Cplant runtime system on a 128 node development cluster. Although some application testing was accomplished, the primary motivation was testing the scalability of initialization. It is our hope to accomplish further testing either at Sandia or in cooperation with another laboratory or interested site.

In the area of systems management and monitoring there is ongoing research into the use of IPMI and development of a portable RAS system at Sandia Labs, funded by the CSRF. It is anticipated that this work will lead to much improved capabilities in this area and provide a compatible environment for the light-os or even a true LWK on cluster systems.

Integration of a scalable parallel file-system, as mentioned previously, would be beneficial for many reasons. We hope to accomplish integration with Lustre, Panasas, and possibly others. We also hope to investigate loading applications along with the light-os package. If this proves to be efficient it could reduce the complexity of runtime software. If not, it may still have utility for systems that run few applications.

8 Acknowledgments

We would like to thank the Computer Science Research Foundation for funding the initial program that allowed this work, and their continued support in many of the related areas mentioned in this paper. We would also like to thank our department manager, James Ang, and our previous manager, Doug Doerfler, for their continued support for work in this area.

A Prototype Implementation of Light-os

This Appendix details a light-os implementation, which was created using standard and freely available Linux tools and software. This implementation was created as a prototype for a more targeted implementation. In so doing, we attempted to create a version of the light-os that could be replicated by someone using freely available tools and resources.

A.1 Overview

This implementation requires a server (admin) node and one client (compute) node. The admin node must be configured as a Dynamic Host Configuration Protocol (DHCP) and Trivial File Transfer Protocol (TFTP) server, since the admin node will provide the kernel and initrd to the compute node over the network. The compute node uses the Pre-boot Execution Environment (PXE) to begin the boot process. It obtains boot information from the DHCP server and its kernel and initrd via TFTP. During the kernel initialization process, the initrd is mounted and the linuxrc program is executed.

At this point, we take advantage of the fact that the linuxrc program can be any valid executable or script. For the light-os implementation, we created a specialized linuxrc script (detailed in in Section A.3) that prepares a tmpfs file-system with only what is necessary for the final root file-system and executes a call to `pivot_root` to switch into that new root filesystem. The novelty of our approach lies in the specialization of the linuxrc script and the use of tmpfs as the final root file-system. In general, the process to create a light-os is as follows:

- Build a kernel
- Create and populate an initrd
- Customize the linuxrc script
- Boot the node
- Compile MPICH

The following sections describe our approach in greater detail[†].

A.2 Build a Kernel

We started with standard Linux kernel source rpms. As a baseline, we started with Fedora Core 3 (kernel-smp-2.6.11-1.27_FC3.x86_64.rpm) and Suse 9.1 (kernel-source-2.6.4-52.x86_64.rpm). We then removed all non-essential kernel support for both devices and services (by non-essential, we mean anything that

[†]The Cluster Integration Toolkit provides a light-os module that automates the steps for building an initrd and customizing your linuxrc script. To download this module, see http://www.cs.sandia.gov/cit/.

is not directly required to support the final system environment). In removing these services and devices, we aimed to minimize the kernel as much as possible. The following section describes some specific kernel options that are required in the kernel configuration file[29]. It is important to note that all of these capabilities were compiled directly into the kernel and not built as loadable modules.

A.2.1 Kernel Details

During initialization the initrd is loaded into ramdisk, which requires the following **Block Device** options:

- `CONFIG_BLK_DEV_RAM`
- `CONFIG_BLK_DEV_INITRD`

Since a DHCP server provides the kernel, initrd, and each node's IP address, we included the following **Networking** options:

- `CONFIG_IP_PNP`
- `CONFIG_IP_PNP_BOOTP`
- `CONFIG_IP_PNP_DHCP`

We enabled **Networking device support** for the specific NIC used by our system:

- `CONFIG_TIGON3`

Since tmpfs is a key part of the light-os environment, we included the following **Filesystems** option:

- `CONFIG_TMPFS`

We used NFS to mount the users' executables on the compute node, so under **Network File Systems** we included:

- `CONFIG_NFS_FS`

We used Dropbear for ssh access, which required two options that are not included as part of the default kernel configuration. The following **Networking Options** had to be included in the kernel:

- `CONFIG_PACKET`

And tty support had to be selected as a **Pseudo Filesystem** option:

- `CONFIG_DEVPTS_FS`

A.3 Create and populate an initrd

After we configured and built the kernel, we created and populated the initrd with the libraries and tools for the final system environment. In particular we used Buildroot[28], which provides a skeletal root file-system, and uClibc[31], a C library alternative to glibc for embedded Linux systems development. Within the Buildroot environment, we included Busybox, which provides a surprisingly extensive suite of standard utilities, but occupies a minimal amount of physical space. Additionally, we included Dropbear for ssh access. In our tests, the final light-os environment was typically smaller than 1 Megabyte.

A.3.1 Details: Buildroot

Buildroot includes a configuration utility, similar to the one for the Linux kernel that supports `make defconfig`, `make config`, and `make menuconfig`. The default configuration for Buildroot includes Busybox, but we had to explicitly select Dropbear from **Package Selection for Target**.

- `BR2_PACKAGE_DROPBEAR`

Because our light-os was a low entropy environment, we compiled Dropbear to use `/dev/urandom` by selecting the following configuration option from **Package Selection for Target**.

- `BR2_PACKAGE_DROPBEAR_URANDOM`

This option prevents Dropbear from blocking while waiting for entropy. While this is a less secure option, since it does not ensure that the ssh host key is sufficiently random, our node was on a private network, and therefore security was less of a concern.

Buildroot constructs the target file-system under the following path: `$BUILDROOT_HOME/build_$ARCH/root/`, where `$ARCH` is the target architecture. When adding or removing options from Busybox, the recommended procedure is to build with the Buildroot default options, run `make menuconfig` in the `build_$ARCH/busybox` directory, copy the new .config file to `package/buildroot/busybox/busybox.config`, and recompile Buildroot. For the light-os, we used the default Busybox configuration options, but explicitly selected the following option from **Linux System Utilities**:

- `CONFIG_FEATURE_MOUNT_NFS`

A.3.2 Details: initrd

The process for creating an initrd for Linux is quite standard and well documented; however, we have reproduced it here to be complete. The one important caveat is that we increased the number of inodes from the default as the light-os requires a lot of inodes relative to its small physical size.

- We created and zeroed out our new initrd file from within the tftp directory using the `dd` command.

  ```
  dd if=/dev/zero of=initrd bs=1024 count=2048
  ```

- We created the file-system, increasing the number of inodes using the `-N` option.

  ```
  mke2fs -F -m 0 -b 1024 -N 1000 initrd
  ```

- We mounted the file-system so we could populate it with the newly built Buildroot environment.

  ```
  mount -t ext2 -o loop initrd initrd_mnt
  ```

- Finally, we populated the file-system. From within our `$BUILDROOT_HOME/build_$ARCH/root/` directory, we executed the following command:

  ```
  find .  -print |cpio -pdmv /initrd_mnt
  ```

A.3.3 Details: the root file-system

Some additional modifications to the root file-system were required in order to boot the node. Buildroot creates the `/dev` entries as regular files, so they must be recreated using the `mknod` command. Secondly, we changed the permissions of the entire root file-system to be owned by root. We also removed the `/etc/mtab` file, and edited the `/etc/fstab` file to include the entries for proc and devpts as follows:

```
# <file system> <mount pt> <type> <options> <dump> <pass>
proc /proc proc defaults 0 0
devpts /dev/pts devpts defaults,gid=5,mode=620 0 0
```

Although Buildroot provides an inittab file, it made certain assumptions that did not suit the light-os environment. For example, we removed any commands associated with logging functionality, which unnecessarily increased the size of our light-os environment. We edited the `/etc/hosts` file to reflect the organization of our network.

The init process is slightly different for Busybox than for standard Linux distributions. Instead of having a variety of run levels with links to scripts, the inittab simply invokes `/etc/init.d/rcS`, which sequentially starts all the scripts in the init.d directory.

To enable ssh keys, we followed a procedure typical for OpenSSH, and then copied the keys into the correct directory within the initrd. We ensured that `~`, `~/.ssh`, and `~/.ssh/authorized_keys` were writable only by the user (or root), otherwise Dropbear will ignore the files.

A.4 Customize the linuxrc script

The linuxrc script orchestrates the configuration of the final system environment. It creates a tmpfs root file system in `/tmpfs_root`, which keeps all files in memory, and has the advantage over RAM disks of being able to dynamically increase and decrease in size. Our customized linuxrc copies `/bin`, `/lib`, `/etc`, `/sbin`, `/usr`, `/root`, and `/var` into `/tmpfs_root`, which will become the new root file-system. It creates any new directories that might be required; in this case, `/proc`, `/initrd` (where our old initrd will be mounted) and `/dev/pts` (for ssh). It also uses mknod to create the necessary devices relative to our new root (`mknod /tmpfs_root/dev/mem c 1 1`)[†]. It then moves into the new root (`cd /tmpfs_root`) where it creates a link, called linuxrc, to the Busybox executable. (Since Busybox is a multi-call binary, by creating a link to Busybox, called linuxrc, we are effectively setting up our environment so that Busybox's default init process can be invoked upon changing into the new root.) Then, the `pivot_root` command is invoked from within the new root, which also mounts the old root on the initrd directory (`/sbin/pivot_root . initrd`). The script unmounts the original initrd after changing into the new root in order to save additional space. As a last step, the script chroots into the new root, launches the new linuxrc, and redirects output to a console:

```
exec /usr/sbin/chroot . /bin/busybox linuxrc dev/console 2>&1.
```

A.5 Boot a node

We edited our pxeconfig on the admin node file to reflect the new method of booting. The following is an example entry in a pxeconfig file used to boot the light-os. The kernel DHCP client provides the compute node with its IP address so that there is no need for networking or configuration scripts within our light-os environment.

```
LABEL light-os
KERNEL /vmlinuz-2.6.8-24-smp
APPEND initrd=/initrd root=/dev/ram0 ip=dhcp rw console=tty0
init=/linuxrc
```

A.6 Compile MPICH

We also added support for MPICH[30], a free implementation of the Message Passing Interface (MPI). To do so, we cross-compiled mpich against the uClibc libraries, since the glibc system calls, `getpwuid` and `gethostbyname`, cannot be statically linked into the user executable. Fortunately, Buildroot provides a cross-compilation toolchain, which enabled us to compile mpich with the uClibc libraries available on our compute node. Mpich also requires either rsh or ssh.

[†]For a complete listing of the necessary devices, see the linuxrc script that is included as part of the light-os CIT module (http://www.cs.sandia.gov) or alternatively, the following website provides an example: http://www.cybcon.com/~coert/linux/siso/kernel-busybox.html.

To accommodate these requirements, we first had to set some environment variables.

- We added the path of the Buildroot cross-compilation toolkit to our `$PATH` environment variable.

  ```
  export PATH=$PATH:/cluster/src/buildroot/build_i686/staging_dir/bin
  ```

- We set our default C compiler to be the uClibc compiler.

  ```
  export CC=/cluster/src/buildroot/build_i686/staging_dir/bin
  /i686-linux-uclibc-gcc
  ```

- We set the `RSHCOMMAND` environment variable to use ssh.

  ```
  export RSHCOMMAND=/usr/bin/ssh
  ```

- We then built mpich with the following commands:

  ```
  ./configure --with-device=ch_p4 --without-romio --disable-sharedlib
  make
  ```

The `--with-device` option specifies that this version of mpich will be compiled to support Ethernet (ch_p4). Since the light-os and the full Linux distribution on the admin node used a different set of libraries, we disabled shared libraries, and for the sake of simplicity, we disabled romio.

After creating a working version of the mpich tools and libraries, we then compiled several test programs, provided with the mpich distribution. We compiled these programs statically, so that they would not rely on shared libraries that might not be available in the light-os environment. Static compilation can produce large executables. The executable size can be reduced using the strip command. In our tests, this typically reduced the file size by about two-thirds.

```
cd $MPICH_HOME/installtest
../bin/mpicpp -static -o cpi cpi.c
strip -o cpi.stripped cpi
```

We executed the test programs with mpirun, which assumes that the executable is available under the same path on all the compute nodes. To satisfy this assumption, we NFS-mounted the directory containing the executable on the light-os node. Although the use of NFS may seem contrary to the aims of the light-os, in this case we used NFS to simulate a high speed file-system, which is typically available in an HPC environment.

References

[1] Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000. Contact: jhlaros@sandia.gov

[2] High Performance Technologies Incorporated, 11955 Freedom Drive, Suite 1100 Reston, VA 20190, Contact: csegura@hpti.com and ndauchy@hpti.com

[3] Linux is the registered trademark of Linus Torvalds in the U.S. and other countries

[4] Kelly, Suzanne M, Ronald B Brightwell, "Software Architecture of the Light Weight Kernel, Catamount," Conference Paper, Cray User Group, May 2005. http://gaston.sandia.gov/cfupload/ccim_pubs_prod/CUG2005-CatamountArchitecture.pdf

[5] Massively Parallel Computing using Commodity Components, Ron Brightwell, Lee Ann Fisk, David S. Greenberg, Tramm Hudson, Mike Levenhagen, Arthur B. Maccabe, Rolf Riesen, Parallel Computing 26 (2000) 243-266. ftp://ftp.cs.sandia.gov/pub/papers/bright/cplant-journal.pdf

[6] Implementing Scalable Disk-less Clusters Using the Network File System (NFS), James H. Laros III and Lee H. Ward, Proceedings of the 4th Symposium of the Los Alamos Computer Science Institute: LACSI 2003, 27-29, October 2003. http://www.cs.sandia.gov/cit/publications/index.html

[7] An Extensible, Portable, Scalable, Cluster Management Software Architecture. James H. Laros III, Lee Ward, Nathan W. Dauchy, Ron Brightwell, Trammell Hudson, Ruth Klundt, Proceedings of the 2002 IEEE International Conference on Cluster Computing, 23-26 Sept. 2002. http://www.cs.sandia.gov/cit/publications/index.html

[8] The Cluster Integration Toolkit - An Extensible, Portable, Scalable Cluster Management Software Implementation. James H. Laros III, Lee Ward, Nathan W. Dauchy, James Vasak, Ruth Klundt, Glen Laguna, Marcus Epperson, Jon R. Stearley Proceedings of the 1st Cluster World Conference and Expo 23-26 June, 2003. http://www.cs.sandia.gov/cit/publications/index.html

[9] Computer Science Research Institute at Sandia National Labs, http://www.cs.sandia.gov/CSRI/About_CSRI/AboutCsriSidebar.htm

[10] Warewulf - http://www.warewulf-cluster.org/cgi-bin/trac.cgi

[11] http://www.clustermatic.org/

[12] Los Alamos National Labs, www.lanl.gov

[13] The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q, Fabrizio Patrini, Darren J. Kerbyson, Scott Pakin. www.sc-conference.org/sc2003/paperpdfs/pap301.pdf

[14] www.kernel.org

[15] BusyBox - www.busybox.net

[16] http://www.lustre.com

[17] http://www.parl.clemson.edu/pvfs/

[18] http://www.panasas.com/

[19] http://www.ibrix.com

[20] http://www.clusterfs.com

[21] http://www.cs.sandia.gov/platforms/RedStorm.html

[22] http://www.llnl.gov/asci/platforms/bluegenel/bluegene_home.html

[23] A Software and Hardware Architecture for a Modular, Portable Extensible Reliability Availability and Serviceability System- James H. Laros III, presented at the 2nd Workshop on HIgh Performance Computing Reliability Issues in conjunction with the 12th International Symposium on High Performance Computer Architecture, February 11th 2006.

[24] http://h18004.www1.hp.com/products/servers/management/ilo/

[25] http://www.sun.com/servers/alom.html

[26] http://www1.us.dell.com/content/topics/
global.aspx/power/en/ps2q02_bell?c=us&cs=19&l=en&s=dhs

[27] http://www.intel.com/design/servers/ipmi/index.htm

[28] Download and documentation for Buildroot can be found at http://buildroot.uclibc.org/.

[29] For a more detailed discussion of the requirements for disk-less kernels, consult the diskless.kernel document that is included in the diskless module of the Cluster Integration Toolkit. This can be downloaded of the CIT website: http://www.cs.sandia.gov/cit/.

[30] Message Passing Interface (MPI) standard. (http://www-unix.mcs.anl.gov/mpi/) Download of mpich can be found at (http://www-unix.mcs.anl.gov/mpi/mpich/download.html).

[31] http://www.uclibc.org