The Linux BIOS

Ron Minnich, James Hendricks, Dale Webster
Advanced Computing Lab, Los Alamos National Labs
Los Alamos, New Mexico

August 15, 2000

Abstract

The Linux BIOS replaces the normal BIOS found on PCs, Alphas, and other machines. The BIOS boot and setup is eliminated and replaced by a very simple initialization phase, followed by a gunzip of a Linux kernel. The Linux kernel is then started and from there on the boot proceeds as normal. Current measurements on two mainboards show we can go from a machine power-off state to the "mount root" step in a under a second, depending on the type of hardware in the machine. The actual boot time is difficult to measure accurately at present because it is so small.

As the name implies, the LinuxBIOS is primarily Linux. Linux needs a small number of patches to handle uninitialized hardware: about 10 lines of patches so far. Other than that it is an off-the-shelf 2.3.99-pre5 kernel. The LinuxBIOS startup code is about 500 lines of assembly and 1500 lines of C.

The Linux BIOS can boot other kernels; it can use the LOBOS(ref) or bootimg(ref) tools for this purpose. Because we are using Linux the boot mechanism can be very flexible. We can boot over standard Ethernet, or over other interconnects such as Myrinet, Quadrix, or Scaleable Coherent Interface. We can use SSH connections to load the kernel, or use InterMezzo or NFS. Using a real operating system to boot another operating system provides much greater flexibility than using a simple netboot program or BIOS such as PXE.

LinuxBIOS currently boots from power-off to multi-user login on two mainboards, the Intel L440GX+ and the Procomm PSBT1. We are currently working with industrial partners (Dell, Compaq, SiS, and VIA) to port the LinuxBIOS to other machines. According to one vendor, weshould be able to purchase their LinuxBIOS-based mainboards by the end of this year.

1 Introduction

Current PC and Alpha cluster nodes, as delivered, are dependent on a vendor-supplied BIOS for booting. The BIOS in turn relies on inherently unreliable devices – floppy disks and hard disks – to boot the operating system. While there has been some movement toward a net booting standard, the basic design of the netboot as defined by PXE (ref) is inherently flawed, and not usable in a large-scale cluster environment.

Further, while the BIOS is only required to do a few very simple things, it usually does not even do those well. We have found BIOSes that configure all mainboard devices to a single interrupt (from Compaq); BIOSes that do not configure memory-addressable cards to page-aligned addresses(Gateway and others); BIOSes that will not boot without a keyboard attached to the machine (some versions of the Alpha BIOSes); BIOSes that will not boot if the real-time-clock is invalid (Alpha BIOSes). One engineer has reported to us that he had to write his own PCI configuration code because no BIOS extant could handle his machine's bus structure. BIOSes also routinely zero all main memory when they start, which makes finding lockups in operating systems very hard, since on restart the message buffer and the OS image is wiped out.

Another problem with BIOSes is their inability to accomodate non-standard hardware. For example, the Information Sciences Institute (ISI) SLAAC1 reconfigurable computing board will not work with some BIOSes because it does not configure its PCI interface quickly enough. In some cases, on some mainboards, by the time the BIOS is done probing the PCI bus the SLAAC1 card is not even ready to be probed. As a result the BIOS never finds the SLAAC1. Changing the SLAAC1 hardware is not an option, because the speed problem is an artifact of the FPGA that is used for the PCI interface. If we had control over the BIOS, however, we could change it to accomodate experimental boards which do not always work with standard BIOSes.

Finally, BIOS maintenance is a nightmare. All Pentium BIOSes are maintained via DOS programs, and configuration settings for most Pentium BIOSes and all Alpha BIOSes is via keyboard and display (and, in some truly deranged cases, a mouse). It is completely impractical to walk up to 1024 racked PC nodes and boot DOS on each one in turn to change one BIOS setting. Yet this impractical method is the only one supported by vendors.

This situation is completely unacceptable for clusters of 64, 128, or more nodes. The only practical way to maintain a BIOS in a cluster is via the network, under control of an operating system: BIOS configuration and upgrade should be possible from user programs running under an operating system. BIOS parameters should be available via /proc.

2 Related Work

There are a number of efforts ongoing to replace the BIOS. For the most part these efforts are aimed developing an open-source replacement for the BIOS that supports all BIOS functions – in other words, plug-compatible BIOS software. An end goal of most of these efforts is to be able to boot DOS and have it work correctly. Example projects are the OpenBIOS (www.openbios.org) and the FreeBIOS (which can be found at www.sourceforge.net).

Other efforts aim to build a completely open source version of the IEEE Open Boot standard. This work will require the construction of a Forth interpreter; code to support protocol stacks such as TCP/IP; other code to support Disk I/O and file systems on those disks; in short, many pieces that are already available in real operating systems. If the experience of other similar projects are any indication, this BIOS will take almost as much NVRAM as a gzip-ed Linux kernel, while having much less capability. As an example, the Intel BIOS for the L440GX+ mainboard takes 50% more memory (750 KB) than the LinuxBIOS, and is far less capable.

Finally there is one recent effort which is aimed at providing a commercial replacement for the BIOS, called SmartFirmware (http://www.codegen.com/SmartFirmware/index.html). SmartFirmware is also IEEE Open Boot compliant.

While these efforts are interesting they do not address our needs for clustering. The problem with the BIOS is not only that it is closed-source; the problem is that it is performing a function we no longer need (supporting DOS), rather than functions we do need. The limited nature of the BIOS was originally imposed in the days of 8 KBYTE EPROMS; now, given that the BIOS is using most of a 1 MBYTE NVRAM, we ought to be able to do better. The next generation of mainboard NVRAMs will be 2 Mbytes: there is lots of room for a real operating system on the mainboard. Finally, we need to have the option of doing things the BIOS writers can not imagine, such as booting over Scaleable Coherent Interface or managing the BIOS via SSH connections.

3 Structure of the Linux BIOS

The structure of the LinuxBIOS is driven by our desire to avoid writing new code. We want to build the absolute minimum amount of code needed to get Linux up, and then let Linux do the rest of the work. Linux has shown its ability to handle quirky hardware; it is doubtful we can do a better job than it can.

One initial concern for any PC BIOS is what mode to run the processor in. All x86 family processors boot up into an 8086 emulation mode, running with 16-bit addresses, operators, and operands. In other words, 1000 Mhz. Pentiums on startup are emulating a 16-bit, 6 Mhz, near 25-year-old processor. BIOSes even now have to take into account such unpleasant details as near and far pointers (see the PXE standard for some examples).

Some of the new open source BIOS efforts have taken the approach of doing a fair amount of startup in 16-bit assembly, and at some point making a transition to protected mode. There are three serious problems with this approach. First, the 8086 emulation is guaranteed to work for DOS, but we have seen cases where certain instruction sequences don't seem to work right. Emulation has its limitations, and depending on it seems very risky. Second, some hardware initialization absolutely requires having access to 32-bit addresses. For example, setting up PC-100 SDRAM requires at some point a write to all the memory SIMMs, which are 256 Mbytes in some cases. This addressing requires 30-bit addresses to cover 1 Gbyte of RAM. These addresses are not accessible in 16-bit mode, requiring extraordinarily difficult initialization sequences. Finally, use of 16-bit code requires use of a 16-bit assember such as NASM or as86. Having any assembly code is bad, but having two different kinds of assembly code and two assemblers is unacceptable.

Our decision wat to have LinuxBIOS make a transition to 32-bit mode immediately. The transition is actually a fairly simple process: the processor has to load a segment descriptor table (the so-called "global descriptor table" – GDT) and enable memory protection. To load the GDT the processor must execute an LGDT instruction, and supply a pointer to a table descriptor in addressable memory. Fortunately there is no problem with having this table in NVRAM, so moving to protected mode in the first few instructions is no problem. In fact the sequence takes about 10 instructions.

Once the processor is in 32-bit mode it must do basic chipset initialization. While one might expect that chipset initialization would require reams of assembly code, in actuality assembly code is only needed to turn DRAM on. Once DRAM is on, C code can be used. The advantage of using C, besides the obvious improvement in productivity, is that non-Pentium mainboards have a lot in common with Pentium mainboards, and in some cases even use the same chipsets. LinuxBIOS code can be portable between these different mainboards.

Another potential problem is that LinuxBIOS boots and runs on hardware that is completely uninitialized. LinuxBIOS can not make any assumptions about the state of any hardware; the hardware is in an indeterminate state. Linux, on the other hand, assumes that all hardware is initialized by the BIOS. While most Linux hardware initialization still works with uninitialized hardware, we are also finding that we have had to make minor changes to the kernel. For example, Linux assumes that if an IDE controller is not enabled, it is because the BIOS disabled it. On uninitialized hardware, however, the IDE controller is not enabled because there is no BIOS to enable it in the first place. Thus, 'IDE controller not enabled' has a diametrically opposite meaning in BIOS and non-BIOS environments. We have chosen to make the one-line change to the Linux IDE driver to enable IDE controllers when they are found. The change is controlled by an #ifdef LINUXBIOS.

In the end, the structure we arrived at is very simple. There are five major components: protected mode setup; DRAM setup; transition to C; mainboard fixup; and kernel unzip and jump to kernel. We cover these components below.

3.1 Protected mode setup.

Protected mode setup is 17 assembly instructions that put the segmentation, paging and TLB hardware in a sane state and then turns on protected mode (segmentation, not paging). It operates as follows:

1. First instruction, executed in 16-bit mode at address 0xffff0: jump to BIOS startup. This jump is a standard part of x86 processor reset handling.

2. Next five instructions: disable interrupts, clear the TLB, set code and data segments registers to known values.

3. Next instruction: load a pointer to a Global Descriptor Table (GDT). The GDT is a control table for managing addressing in segmented mode.

4. Next four instructions: turn on memory protection

5. Next several instructions: do remaining segment register setup for protected mode

6. At this point, 17 instructions in, we are in protected mode, can address 4 GB of memory, and are running as a Pentium, not an 8086.

3.2 DRAM setup.

DRAM on current mainboards often requires some level of configuration. SDRAM can be particularly tricky. Since there is no working memory hardware initially, this code is assembly. This code has proven to be the hardest code to get working, consuming several weeks on each mainboard. But as we build experience it has also gotten easier to figure out. We resolved several L440GX+ bugs after getting the code working on the Procomm BST1B. This code takes 79 lines of assembly on the Intel and 280 lines of assembly on the Procomm. These numbers will change somewhat as we plan to modify the Intel support to use the Serial Presence Detect capability on the SDRAM.

3.3 Transition to C.

The rest of LinuxBIOS is written in C, so the next few instructions set up the stack and call a function to do the remaining hardware fixup.

3.4 Mainboard fixup.

Mainboard fixup involves limited hardware initialization that we need to finish loading the kernel. In the current implementation, this includes:

- doing whatever the mainboard requires to turn on caching (setting up an MTRR on some processors) so that kernel unzip runs in reasonable time. If the MTRR is not on, the unzip stage can take one minute. With caching on it is not easily measured.
- Making the full FLASH memory available to the processor. Most standard chipsets come up with only 64K or 128K of the FLASH addressable. Enabling all of FLASH requires some register manipulations, which are different on each chipset.
- Enabling minimum device capabilities in the power management hardware (if there is any).

We also need cover things that Linux can't (or won't) do. Linux can not yet initialize a completely uninitialized PCI bus, although it is getting close. We do the bare minimum by setting the Base Address Registers to reasonable values, and Linux properly handles the rest of the work, such as setting up interrupts and turning off the option ROMs. Linux also needs the keyboard to be in some sort of reasonable state, so we reset it. Finally, clock interrupts have to be working, so we make sure these are turned on. This last detail is mainboard-dependent to a limited extent.

Mainboard fixup is currently 330 lines of C code.

3.5 Inflate the kernel.

This is fairly standard code grabbed from the Linux kernel. It has been extended in a few ways. First, parameters from LinuxBIOS to the Linux kernel proper are handled in this code. The parameters are copied to the standard location for Linux kernels for the given architecture. The command line is also copied out. Also, the standard Linux gunzip won't work in a ROM-based environment without some changes. For the most part these involve declaring initialized arrays as 'const' so that they are placed in the read-only text segment (i.e. FLASH) instead of RAM. Initialized automatic and global variables must also be changed so that they are initialized at runtime.

Once parameters (such as memory size) and cmdline are copied out, the standard Linux kernel gunzip is called and the kernel is unzipped to the standard location for the architecture (0x100000 on PCs).

3.6 Jump to the kernel.

This is a simple jump instruction. We jump directly to the start of the kernel, since much of the standard Linux kernel setup code is not needed, including the part of the kernel that uncompresses the rest of the kernel; that work is done in the LinuxBIOS startup code. Instead of jumping to the boot setup code as LILO does, we directly enter the kernel in the startup_32 function.

3.7 Summary

The structure of the LinuxBIOS is simple. There is just enough assembly to get things going, and the rest is C. There is just enough C to get the hardware going, and the rest is Linux. For example, we only initialize enough of the MTRR registers (one in most cases) so that the LinuxBIOS code is cached. Linux redoes MTRR initialization anyway: there is no point in doing more than the bare minimum. We do very little PCI configuration, in fact just enough so that Linux can do the rest. Also, LinuxBIOS is configured at build time (somewhat as the BSD kernel is) so that it is configured to the mainboard it is built for. The only code in the mainboard NVRAM is the code needed for that mainboard.

We have been told that 100% of common PC BIOS code is assembly. If we count Linux, then something like 1% of LinuxBIOS is assembly. Because so much of the standard PC BIOS is object code, a lot of code in the standard PC BIOS is directed to figuring out what the hardware is. We do not have this problem, since we are source-based.

4 Status

LinuxBIOS is currently booting from power-up on two mainboards. The first is the Intel L440GX+, and the second is the PROCOMM BST1B. We have just begun work as of June on the Compaq DS10 (Alpha) and Dell 2450 (dual Pentium) mainboards. VIA is sending us a motherboard based on their chipset and we expect to start that work in July.

We generally hear two major concerns about LinuxBIOS. The first is that it is going to be impossible for us to track and support all the various mainboard chipsets in use, which in turn will make coverage of 100% of the mainboards impossible. The second is that less than 100% coverage is equivalent to failure.

It is true that the first two mainboards took a lot of work – about four months of one person for both mainboards. Much of this work was non-recurring, as it involved working out the structure of LinuxBIOS and learning about such things as the quirks of SDRAM and the limits of Linux PCI initialization. We hope to have a better estimate of effort involved once we have completed the next two or three mainboards.

While it is true that we can not hope to cover every mainboard in existence, we do not plan to. Our long-term goal in this effort is to have the vendors pick up the porting effort. One (SiS) is already devoting substantial personnel effort to supporting their new 630 chipset. Other vendors are also interested, once we have developed an initial proof-of-concept for one of their mainboards. One or two other vendors are discussing the construction of mainboards designed from the start to run only LinuxBIOS.

The second concern is coverage. If we do not cover 100% of the mainboard market, have we failed?

We would argue that this question is ill-posed for two reasons. First, coverage at an instant in time says nothing about eventual coverage. In 1991, Linux and the BSD operating systems initially only covered a very small fraction of mainboards extant. Many argued that these systems could never succeed, since they could never support all the hardware available. Linux and the BSDs are very successful, and they still do not cover 100% of the mainboards for sale.

Second, 100% coverage is not necessary for success. If LinuxBIOS is available on a reasonably large number of mainboards, and we can buy those boards for clusters, then for our purposes we have succeeded: the market for LinuxBIOS mainboards is economically viable, and we can buy what we need. Our goal is not "LinuxBIOS on every desktop". Our goal is to be able to buy LinuxBIOS-based mainboards from which to build clusters and other computer systems. Economic viability in this market doesn't require 100% mainboard market penetration.

5 Post-startup Scenarios

In this section we describe some possible uses of the Linux BIOS once it is booted. These uses include a network boot; standard boot; a diskless node boot; and maintenance activities. There are also some unique startup modes that have not been tried to date for clusters. In each case, the Linux kernel can boot another Linux kernel using the LOBOS system call we describe in a companion paper.

5.1 Network boot

Once the BIOS has booted Linux from NVRAM, the kernel take can a number of actions. One would be to establish an SSH connection to a remote DHCP configuration daemon. Using SSH represents a significant advance over the UDP-based approach used by, e.g., bootp and PXE. The SSH-based connection can be much more secure. The SSH connection also benefits from the more stable behavior of TCP under load, as compared to a UDP-based approach. We have seen cases where 32 nodes try to netboot off of one server and only 20 succeed. PXE would have severe problems in this case as the PXE standard suggests that a node only send four packets, then give up.

The DHCP daemon can tell the node its identity and direct it as to which kernel to boot. The DHCP server can even send a kernel image over the SSH connection, and the kernel can boot it using LOBOS.

In order for the kernel to use SSH, we need basic functions that are currently buried in the various SSH client programs. To address this problem we are building a library, based on the OpenSSH source, that will allow both kernels and user programs to create connections to SSH daemons and establish encrypted TCP connections. We may also see if CIPE is useful for this purpose.

5.2 Standard boot

The DHCP daemon could send the kernel its parameters and then tell it to continue booting. In this case, the NVRAM kernel is the kernel of choice; no further kernel loading is needed. Or the kernel could boot another kernel off a local disk or other file system resource such as CDROM. This standard boot is almost identical to what is done on cluster nodes now save for two crucial differences:

1. The boot sequence is entirely under the control of a remote node. The boot will be about as fast, but there is much better control.

2. All existing boot sequences for cluster nodes rely on devices with moving parts, either floppy disks, CDROMs, or hard disks. Our LinuxBIOS-based boot sequence relies on devices with no moving parts, namely the NVRAM on the mainboard. If the node is told to use a hard drive and the hard drive has failed, the node can report the failure to the control node. No more guessing when a cluster node won't come up!

5.3 Diskless node

The DHCP daemon could direct the kernel to mount file systems via NFS (no problem since this is a full-featured kernel), AFS, Coda, InterMezzo, or any other network file system. The kernel could then proceed or boot a different kernel via the network file system being used. The kernel can also use many different transports for the network flie system, such as MyriNet, GigaNET, and SCI. We have worked with Peter Braam to enable InterMezzo to cache an entire root file system, so one option is to cache the root file system to a RAM disk. Experiments have shown that a capable root file system can be contained in a 256 MByte RAM disk. Since InterMezzo supports fetch-on-demand, initially only about 50 MBytes of files need to be loaded.

5.4 Maintenance

The DHCP daemon could also direct the kernel to make an SSH port available for remote maintenance. The node would thus not even start /sbin/init, and would instead wait for instructions from a remote maintenance control program. Instructions could include changing LinuxBIOS parameters or even writing a new test kernel to NVRAM. The kernel could even load a new root file system or repartition the local disk. This maintenance model represents a major advance over what we have now.

Using Linux to write to the NVRAM is tricky. If the write fails for any reason there needs to be a way to recover. For now, we are depending on the BIOS recovery NVRAM. In future, and as the on-board NVRAM grows in size, we expect to have two kernels in the NVRAM, a recovery kernel and a normal boot kernel. The bootstrap code will decide which one to run. If the recovery kernel is damaged in some way the bootstrap code can run the recovery kernel.

Another possibility is to have the kernel open a port and communicate using HTTP commands. We could do BIOS maintenance with a web browser.

5.5 Netboot over Myrinet

All PC netboot standards extant (including PXE) rely on a netboot ROM running in 16-bit mode for network packet I/O. Needless to say this requirement greatly reduces the number of interfaces we can use. Since LinuxBIOS boots a true Linux kernel, we can use high performance network interfaces such as Myrinet for the netboot process. Loading a new disk image over Myrinet would similarly be much faster than Ethernet-based disk image loading.

5.6 Netboot over Scaleable Coherent Interface (SCI)

This model is perhaps the most interesting. SCI is a memory-based network, and moving data requires only a bcopy. When the kernel comes up it could configure an SCI interface. Once the kernel has the interface configured it could use SCI (via a fetch-and-add operation which only takes 6 microseconds) to notify a remote server. The remote server can bcopy a kernel image to the local node, and the local node can use LOBOS to boot this image. Or, more interesting, the remote server can bcopy a whole ramdisk image in, and the local kernel can use that image. SCI bandwidth on 32-bit PCI is about 80 MBytes/second, so moving over a RAMDISK image could be done in a few seconds. SCI has the further advantage that configuring the interface only requires 128 bytes of configuration data, and in fact the interface can be configured from a remote controller node. LinuxBIOS doesn't have to load microcode into the interface, as it does for Myrinet.

5.7 Netboot using IP Multicast

If the whole cluster is booting, we can use IP Multicast to distribute kernel and disk images. Most current tools that support this mode (e.g. Ghost) run under DOS. In our case, we have a full Linux kernel at our disposal and can use IP Multicast for most data, and open TCP sockets as needed to recover lost packets.

6 Current Status

LinuxBIOS is currently booting from power-on to multiuser mode on our Intel L440GX+ and Procomm BST1B mainboards. We are also working with Compaq on the Photon (Pentium) and DS10 (Alpha) systems; Dell on their 2450 server; and VIA on one of their mainboards. The code is available on sourceforget.net.

7 Conclusions

LinuxBIOS is a small BIOS that completely replaces the standard BIOS with a simple bootstrap that loads Linux from NVRAM and starts it up. LinuxBIOS makes new types of cluster configuration, maintenance, and startup models possible that were not practical to date. Based on our near ten years of work with clusters we feel that LinuxBIOS is essential to the construction of maintainable clusters.

LinuxBIOS is working and we are beginning to receive some interest and support from vendors. One vendor has sent a pre-production mainboard and most of their BIOS source. Other vendors are providing information and engineering support, including support for non-Pentium processors. We feel it is only a matter of time before the 16-bit, closed-source model of BIOSes is abandoned in favor of Open Source BIOSes with extended capabilities that the community needs