USENIX Association

Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference

San Antonio, Texas, USA
June 9-14, 2003

Awarded Best Student Paper!

Flexibility in ROM: A Stackable Open Source BIOS

Adam Agnew, Adam Sulmicki, *Ronald Minnich, William Arbaugh
Department of Computer Science
University of Maryland at College Park
*Advanced Computing Lab, Los Alamos National Lab**
Los Alamos, New Mexico
LANL LA-UR-03-2137
agnew@cs.umd.edu, adam@cfar.umd.edu, rminnich@acl.lanl.gov, waa@cs.umd.edu

Abstract

One of the last vestiges of closed source proprietary software in current PCs is the PC BIOS. The BIOS, most always written in assembler, operates mostly in 16 bit mode, and provides services that few modern 32 bit operating systems require. Recognizing this, the LinuxBIOS founders began an effort to place a Linux kernel in the ROM of current motherboards– completely removing the legacy BIOS. While the LinuxBIOS effort fully supports Linux, other modern operating systems, e.g. *BSD, and Windows 2000/XP, could not be directly supported because of their reliance on a few services provided by those legacy BIOSes. In this paper, we describe how we have combined elements of the LinuxBIOS, the Bochs PC emulator, and additional software to create the first open source firmware for the IBM PC capable of booting most modern operating systems.

1 Introduction

The personal computer (PC) basic input output system (BIOS) remains one of the last bastions of closed source software. The reasons for this are many, but the most prominent is that many of the hardware interfaces in modern PC chipsets are covered by non-disclosure agreements. NDAs greatly limit the set of systems that an open source BIOS could support. Additionally, technical barriers to entry at the firmware/BIOS level are significant. For example, debugging is considerably more difficult, and the tools to permit control over the processor at the firmware level, e.g. an in-circuit emulator, are very expensive.

These technical and non-technical barriers, coupled with a near monopoly in the BIOS market, have prevented innovation from occurring in a key element of the personal computer. As a result, commercial BIOSes continue to be written entirely in assembler, run mostly in 16 bit mode, and provide few services beyond the interrupts initially provided by the BIOS in the 1980's.

The LinuxBIOS effort at Los Alamos National Labs sought to change that situation. Frustrated by the problems created by the BIOS in large sets of computing systems, i.e. clusters, the LinuxBIOS team began to create their own open source BIOS. The LinuxBIOS also sought to place an operating system kernel, e.g. Linux, in the ROM of a PC motherboard.

The reason for putting a full kernel in place of the BIOS is simple: Linux does a far better job of detecting and configuring hardware than standard BIOSes do, and its "footprint" in memory is not really that much larger. As of 1999 there are many systems (kmonte, kexec, bootimg, LOBOS) that allow Linux to boot another operating system. The result is that LinuxBIOS can load a kernel over any device, network, protocol, and file system that Linux supports– compelling reasons to use Linux as a boot program.

Even with the considerable technical barriers, LinuxBIOS now runs on numerous motherboards. While it remains unlikely that LinuxBIOS will ever become available on all PC motherboards, the current success of the project is significant.

LinuxBIOS is actually divided into two parts. The first part is a small open-source system startup code. The second part is the Linux kernel itself. Over time, users have found that they can use the first part without the second, and the Linux kernel has been replaced by many different types of software, from Etherboot to 9load, the Plan 9 loader. This has led to a confusing situation with respect to naming: in the early days of LinuxBIOS, the

software implicitly included the Linux kernel. Nowadays, the name "LinuxBIOS" has gradually come to mean only the small open source system startup code. For the rest of the paper, when we use the term "LinuxBIOS" we mean the small startup code, not the full startup code plus Linux kernel combination.

While the original motivation for LinuxBIOS was focused solely on improving the management of large computing clusters, numerous developers viewed LinuxBIOS as a means to provide additional features in the BIOS such as security, and support for additional operating systems beyond Linux. In this paper, we describe one of those efforts– an open source stackable BIOS.

The elements of the stackable BIOS are shown in Figure 1. The first element is LinuxBIOS. LinuxBIOS performs all of the steps necessary to initialize the hardware on the motherboard. The next step is ADLO, the "ADhesive LOader." This is stackable BIOS software specifically written to serve as the "glue" between the LinuxBIOS and the next element, the Bochs BIOS. Bochs is a highly portable open source x86 PC emulator which includes emulation of the x86 CPU and common I/O devices. Bochs also includes a custom BIOS which provides the legacy BIOS functionality needed to boot modern operating systems such as Linux, OpenBSD, and Windows 2000 using the GNU Grub bootloader.

The resulting BIOS which comes from the combination of LinuxBIOS, ADLO, and the Bochs BIOS has proven to be quite slim and fast.

LinuxBIOS, ADLO, BochsBIOS, and even the Video BIOS (typically 64KB) can take up less than 175KB in size (Figure 2). Because alignment data pads out many of the images, a size of less than 100KB can be achieved.

In the remainder of this paper, we present the details of how elements of two relatively mature open source projects (LinuxBIOS and Bochs) were combined with additional GPL'd software written by the authors to completely eliminate the need for a proprietary legacy BIOS.

2 LinuxBIOS

LinuxBIOS is a GPLed project started at the Cluster Research Lab, a division of the Advanced Computing Laboratory (ACL), in Los Alamos National Labs. A computer cluster is a group of computers connected to work together as a parallel computer. The Cluster Research Lab performs research and development in operating systems and cluster design in order to improve the way clusters are built, managed, and used.

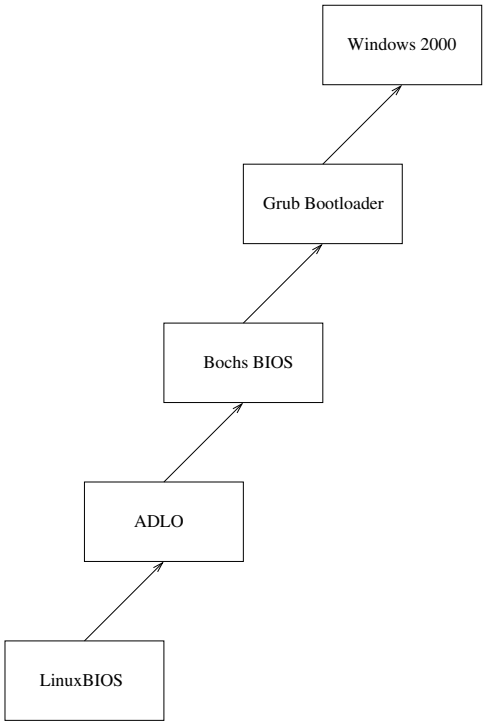

Figure 1: Stackable BIOS Overview

LinuxBIOS was created to fill a need in the Computer Research Lab for an open source BIOS. Currently, x86 motherboards ship with BIOSes supplied by vendors such as Phoenix Technologies and American Megatrends. But these BIOSes had shortcomings which made them impractical for cluster use.

The ACL team found that existing BIOSes do a poor job of setting up a PC for modern OSes. They found:

- BIOSes which configure memory in a suboptimal way. For example, some BIOSes were discovered to use memory capable of CAS2 speeds at a much slower CAS3 setting.
- incorrect configuration of PCI address spaces that open up security holes when memory-mapped cards (e.g. Myrinet) are used.
- suboptimal assignment of interrupt requests. Some BIOSes were found to share IRQs between multiple devices when such sharing was not required. This sharing of IRQs added latency upon interrupts

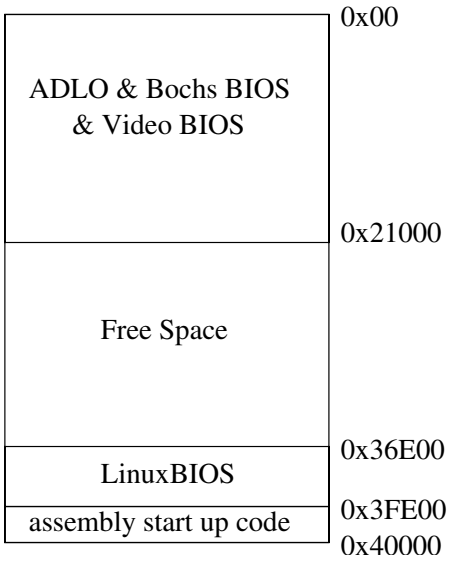

Figure 2: Typical LinuxBIOS + ADLO + BochsBIOS EEPROM map for a 512k part.

needlessly.

- incorrectly configured BIOS tables such as the $PIR table, so that the critical information Linux needs for IRQ assignment is not availble. So pervasive are some of the errors that they have impacted the Linux 2.4.19 kernel; in this version, GEODE IRQ setup was changed, incorrectly, to accommodate some broken motherboards.
- incorrect ID strings in the BIOS, for example the string "Mainboard vendor name here" is in the BIOS instead of the mainboard vendor name
- no way to upgrade the BIOS from a modern OS (some vendors ship DOS diskettes for a BIOS upgrade; others require that you boot a Windows 3.1 CD and run an upgrade program).
- no way to preserve CMOS settings when a BIOS is upgraded (one vendor recommended we record the settings on paper and then re-enter them for each machine; a difficult task on a 960-node system with no keyboard or console).

There are also structural problems with BIOSes that derive from the requirements for supporting DOS. One of these is the requirement that the BIOS zero memory. This requirement makes post-mortem debugging virtually impossible, as a reset results in erasing the memory image of the previously running operating system.

Commercial BIOSes also have problems accommodating custom built hardware, and can take considerable time to boot. Finally, when bugs or errors crop up in a commercial BIOS, it is almost impossible to fix them.

These short-comings made it obvious that an open source BIOS such as LinuxBIOS is needed, especially for the fast paced innovations which can make the next generation of clusters successful.

To accomplish its task, LinuxBIOS is placed in the computer's EEPROM chip (Electronically Erasable Programmable Read Only Memory) where normally a vendor supplied BIOS would reside. LinuxBIOS completely replaces the vendor BIOS; no fragment of the vendor BIOS is left once LinuxBIOS is installed. The LinuxBIOS binary itself is small, typically only 36 kilobytes in size depending on configuration choices. Then, a Linux kernel is placed in the EEPROM chip to act as a bootloader (Figure 3).

The Linux kernel is used for this task because it supports a wide array of hardware that can be used to acquire the binaries necessary for further booting operating systems. The Linux kernel is well suited for this task because it is, for the most part, written to avoid a need for legacy BIOS services. This lack of dependencies reduces the number of services LinuxBIOS must provide and thus results in a small and compact BIOS.

With the complexity and latency of commercial BIOSes removed, the LinuxBIOS developers discovered they were able to accomplish the entire bootstrap process in a matter of seconds.

Pretty soon, the LinuxBIOS developers found there was demand not only from those building clusters, but also from a computer enthusiast community, and even more notably, motherboard vendors who were hopeful of a future where they wouldn't have to pay a royalty to BIOS vendors.

This created several problems for the LinuxBIOS developers if they wished to create an open source BIOS for a wider audience.

- Most motherboards ship with 256Kbyte EEPROM parts. Even under extreme compression, the Linux kernel could not fit in such a small flash.
- LinuxBIOS was kept small and fast by not supporting legacy PC BIOS services. The Linux Kernel used with LinuxBIOS had to be slightly modified

to avoid using these legacy services. Unfortunately, many of the other operating systems that motherboard vendors and computer enthusiasts want to run require these services. It is not practical to modify the BIOS interface of each of these operating systems. This is especially true for operating systems which are closed source.

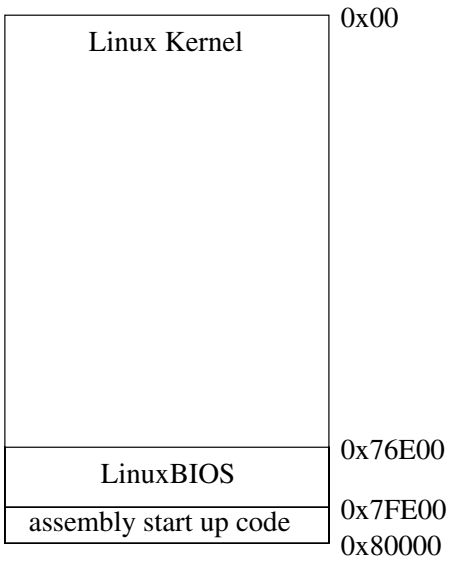

Figure 3: Typical LinuxBIOS EEPROM map (using a Linux Kernel) for a 512k part.

3 Bochs

We were able to solve both of these problems by taking advantage of another open source project, Bochs.

Bochs is an LGPLed project, sponsored by MandrakeSoft, which serves as a highly portable x86 emulator written in C++. Bochs is a true, complete emulator in that it fully emulates an x86 CPU, common I/O devices such as floppy and hard drives, and an x86 BIOS.

The component of Bochs which we found useful in our plans to add PC BIOS services to LinuxBIOS was its BIOS. The Bochs BIOS had many attributes which made it especially appealing for our needs.

- None of the hardware emulated by Bochs is implemented in their BIOS layer. The BIOS was designed to interact with the hardware through true device calls and aimed to conform to up to date standards. This was advantageous to us in that few modifications were needed for the Bochs BIOS to utilize the real hardware on our target platform.
- Since the hardware is emulated, it does not need to be "turned on" like on a real x86 platform. This means Bochs does not implement the process of initializing such things as CPU cache, IDE controllers, etc. These services are already provided by LinuxBIOS for the motherboards it supports.

Further, these attributes meant that the modifications we found necessary to make to Bochs were prime candidates for inclusion back into the Bochs source. All the changes we needed to make to Bochs were merely more correct implementations of the BIOS interrupts it provided. These modifications did not break the Bochs BIOS' compatability with their emulated hardware, and allowed it to work more correctly with the real hardware on our platform as fewer assumptions were made.

This means that further maintenance of the BIOS layer of our solution can stay within the Bochs source and LinuxBIOS can continue to leverage itself effortlessly off of the further hard work of the Bochs PC emulator developers.

4 BIOS Dependancies in Modern Operating Systems

The BIOS services required to boot some modern operating systems are plentiful. Most are trivial and hardly worth mentioning, such as probing for keyboard presence or the timer interrupt. But there are four main services which proved critical to booting both operating systems like Windows 2000 and the Linux Kernel 2.4 series. These services comprised the video BIOS functions, hard drive services, memory sizing, and providing a PCI table. Many more services are useful, but they are still new standards which are optional to use in modern operating systems, see Table 1.

The following section outlines the steps taken to make ADLO and Bochs BIOS functionality closer to that of standard commercial BIOSes and distinguish this functionality from that of its base LinuxBIOS, see Table 2.

4.1 Video BIOS Functions

Among the first of the issues dealt with was providing Video BIOS functions. In a stock LinuxBIOS setup, the issue is averted by rerouting the console output to a serial

BIOS Feature Needed	*Standard 2.4.x Linux Kernel*	*Modified for LinuxBIOS 2.4.x Linux Kernel*	*Windows 2000[3]*	*Windows XP[4]*	*FreeBSD[5]*
Int13 Handling	yes	no	yes	yes	yes
Int15 ax=E820 Map*	yes	no	yes	yes	yes
PnPBIOS	no	no	no	no	no
PCI Table	yes	yes	yes	yes	yes
Video BIOS	no	no	yes	yes	yes
ACPI	no	no	no	no	no
APM	no	no	no	no	no

Table 1: Operating system dependencies on BIOS interrupt functionality.
* While Int15 function ax=E820 is not itself needed, it is the most modern and popular of the Int15 memory sizing functions of which at least one is needed.

Feature	*Standard PC BIOS[2]*	*LinuxBIOS using Linux Kernel*	*LinuxBIOS using Etherboot*	*LinuxBIOS using ADLO and Bochs BIOS*
Int13 Handling	yes	no	no	yes
Int15 ax=E820 Map	yes	no	no	yes
PnPBIOS	yes	no	no	no
PCI Table	yes	yes	yes	yes
Video BIOS	yes	yes	yes	yes
ACPI	yes	no	no	no
APM	yes	no	no	no

Table 2: Features provided by a variety of BIOS configurations

console. Regrettably, this luxury was rarely available in other operating systems, so the issue had to be resolved in another way.

In most commercial BIOSes, Video BIOS functionality is provided by an add-in card containing an expansion ROM. This ROM is found while probing the IO buses for expansion ROMS. When a Video BIOS is found, it is copied into memory area 0xC0000 to 0xC7FFF and executed. The Video BIOS in turn claims control of Int10.

On the motherboard we developed with, however, the video chipset was integrated and the Video BIOS was stored with the commercial BIOS on the motherboard's EEPROM. As these images are to replaced with our code, the Video ROM had to be acquired and loaded in a different fashion.

We accomplished this task by extracting the Video BIOS from memory before installing LinuxBIOS. Since we can expect the Video BIOS to be stored at address 0xC00000, we can boot into Linux using the commercial BIOS shipped with the motherboard and extract this image using 'dd' and the /proc/kcore memory interface. The total size of the image is determined by the 3rd byte, which represents the number of 512 byte blocks. Because the /proc/kcore interface is an ELF image, an offset must be taken into account in order to skip the ELF header. This extraction method will work easily on most integrated and non integrated motherboards alike. Once the video BIOS image is extracted it is stored in the EEPROM, see Figure 2.

Once the Video BIOS is in place, both the Linux and Windows operating systems will use it for such functions as preliminary output when booting, VESA compliance probing, and setting video modes.

4.2 Hard Drive Services and Interrupt 13

Historically, hard drive services have been the most complex and frustrating of the BIOS services for both the BIOS and operating system developers. As hard drives have grown in capacity, they have consistently been the first of the devices to feel the strain imposed by the instruction width of architectures. As a result, there have been several evolutionary steps in addressing modes and the complexity of the Int13 services has become staggering.

Speed constraints, buggy BIOSes, addressing mode revisions, and an increasing variety of devices has compelled designers of modern operating systems to remove

dependency on Int13 for hard drive functions for the most part. Instead, they communicate with the drive controllers directly, in "polled I/O mode", and only the bootloaders, which are constrained by space from having drivers for a variety of controllers, still depend on these services heavily.

In this regard, a LinuxBIOS configuration with the Linux Kernel housed in ROM with LinuxBIOS is again at an advantage. The Linux Kernel has all the necessary drivers for properly communicating with a variety of drive controllers. The only issue that needs to be addressed then is waiting for the hard drives to spin up to operational speed. While the boot times of most commercial BIOSes are long enough that it is taken for granted that hard drives will be fully spun up and available before they are needed, the fast boot times of LinuxBIOS makes this assumption incorrect. To correct the mistake, the kernel is patched to assume drives are present before they are available for actual I/O.

As we aimed to boot unmodified operating systems and bootloaders such as LILO, GRUB, and the Windows 2000 Bootloader, we needed to support a much larger set of Int13 functionality.

The Bochs BIOS provided an excellent starting point toward this end; it implemented virtual devices and defined the Int13 functions necessary for bootloaders and operating systems to interface with them. When Bochs virtualizes these devices under a host operating system, the host operating systems already deals with timing issues with the actual physical devices. For this reason, timing with Int13 had not been taken into account directly in the Bochs BIOS. As our BIOS does not include such a virtualization, we implemented proper handling of timing issues to avoid such complications as trying to access devices before they were spun up to operating speed or reading buffers before they could be filled by the hard drive.

4.3 Memory Sizing

Like hard drive addressing modes, memory sizing has also gone through evolutionary steps as instruction widths and the cost of memory has dramatically changed. While memory sizing used to be accomplished by simply returning the amount of 1K size blocks available through Int15 function ah=088h, limiting the returned amount to 64MB, the service is now commonly performed through subsequent calls to Int15 function ax=0e820h in order to fill sections of a table[8].

A typical Int15 ax=E820 memory table contains several entries, each describing a range in memory with unique characteristics, see Table 3. In this example, the first range identifies the memory from the beginning to 640K as available in keeping with the convention of Lower Memory. Next, a range up to the first megabyte is reserved for BIOS use. The other segments map the rest of available memory, reserving some sections for ACPI use.

An operating system is only interested in the memory ranges which it can use, see table 4. To reduce complexity, we removed sections from the table which were reserved. LinuxBIOS coupled with the Bochs BIOS does not provide ACPI functions, so those sections do not need to be explicitly identified either. The simplified Int15 ax=E820 table was able to get us past issues in development that dealt with an AMI compatible CMOS which we were unable to provide.

4.4 PCI Tables

The original PCI design for interrupts was very simple. PCI supports four interrupts lines, A, B, C, and D. The B, C, and D lines are reserved for multifunction cards, so that only the A line can be used for single-function cards. The problem is that most cards are single-function; most cards drive the A line. Most cards would have to share the A interrupt, leading to high interrupt latency.

This is the reason for one of the more distressing kludges related to PCI busses. Vendors decided to remap the A, B, C, and D lines on the motherboard so as to more evenly distribute the interrupt load. The A line for a given slot may actually be mapped to the B, C, or D interrupt pins on the interrupt controller.

How can an operating system tell, from reading the PCI bus configuration, how the lines are actually mapped? In practice, it can not. Hence the need for the PCI tables.

There are two types of PCI tables: the older $PIR table, and the newer SMP table. The BIOS must supply both these tables, the older one for older operating systems or newer OSes configured to use the old table (e.g. Linux in uniprocessor mode); and the newer one which is required for correct functioning of an SMP OS.

Each table has basic information about the motherboard, including a variable-length list of PCI slots and the mapping of the four PCI slot interrupt lines to the actual interrupt lines on the interrupt controller. This information can be used to determine, for a given interrupt line,

BIOS-e820:	0000000000000000	-	00000000000A0000	(usable)
BIOS-e820:	00000000000F0000	-	0000000000100000	(reserved)
BIOS-e820:	0000000000100000	-	000000001FFF0000	(usable)
BIOS-e820:	000000001FFF0000	-	000000001FFF3000	(ACPI NVS)
BIOS-e820:	000000001FFF3000	-	0000000020000000	(ACPI data)
BIOS-e820:	00000000FFFF0000	-	0000000100000000	(reserved)

Table 3: Int15 ax=E820 memory map in a commercial BIOS

BIOS-e820:	0000000000000000	-	00000000000A0000	(usable)
BIOS-e820:	0000000000100000	-	000000001FFF0000	(usable)

Table 4: Int15 ax=E820 memory map in LinuxBIOS using Bochs BIOS

which card or cards is the source of the interrupt. The operating system has to build an internal mapping of the card IRQs so that it can make this determination.

One of the more troublesome aspects of the tables is that they contain errors. For example, some Geode motherboards have the low order bit set incorrectly. This led to an incorrect patch being applied to the Linux 2.4.19 release kernel which broke the kernel on correct motherboards so the kernel would work on incorrect motherboards. As of this writing this mistake has not been corrected.

4.5 Future Work in ADLO and Bochs BIOS

In recent years, new standards have emerged and been quickly adopted by both BIOS manufactures and the developers of operating systems.

The Advanced Configuration and Power Interface (ACPI) is a particularly powerful specification that aims to take over the services performed by PnPBIOSes, multiprocessor tables, and Advanced Power Management (APM). However, support for ACPI has just recently been started in the Linux Kernel 2.5 series. Many Windows varieties already extensively make use of ACPI when available, but it is still not necessary for booting these operating systems. Int15 function ax=E820 is the first of the services from the ACPI specification which we have chosen to implement and a full implementation of ACPI would be a worthy goal in further development of ADLO and Bochs BIOS.

As ACPI aims to replace the functionality of APM, PnPBIOS, and multiprocessor tables, adding support in ADLO and Bochs BIOS to support these functions would equate to adding a layer of services which are not necessary now and will become legacy in the near future.

Other directions ADLO and Bochs BIOS development will likely take is full support for multiprocessor systems (this has been added into the Bochs BIOS for use in Bochs, but remains to be tested on real hardware), and a virtual Video BIOS to allow for serial consoles like on many server class motherboards.

5 The Bootstrap Process

Though LinuxBIOS and the Bochs BIOS are an open source take on the PC BIOS, neither intended to be a drop in replacement for commercial BIOSes. By stacking and executing the LinuxBIOS, ADLO, and Bochs BIOS components in order, the foundation is in place for an open source BIOS to finally be a viable alternative to commercial BIOSes on many x86 PCs. A description of the tasks performed by each component in this advantageous configuration follows.

5.1 LinuxBIOS

The guiding philosophy of LinuxBIOS is simple: “Let the Linux kernel do it.” In other words, if Linux can perform some task such as device initialization, there is no need for any other software to do that work. We have found that in most cases Linux will redo work that the BIOS did anyway; or, still worse, Linux has to redo work that the BIOS did since in so many cases the BIOS gets it wrong. The BIOS should only do the bare minimum of work that Linux can’t do. By doing the bare minimum, LinuxBIOS becomes fast and small.

In the procedure of booting up the computer to a usable state, LinuxBIOS performs many tasks. First among these tasks is to set up and initialize memory; and set up the serial consoles so debugging information is available. The resources which the LinuxBIOS developers

have available to them on the chipsets for which they develop are often vague, scant, or simply non-existent. With this in mind, it is fitting for a serial console to be such a high priority.

Once the DRAM is initialized, LinuxBIOS can continue in C code. This is a welcome feature when faced with thousands of lines of x86 assembly.

From this point, LinuxBIOS can set up Memory Type Range Registers (MTRRs) on the CPU(s), making the cache of the CPU(s) available and thus speeding up execution considerably. LinuxBIOS is also be responsible for creating an IRQ routing table, and initializing individual hardware components on the motherboard. These often include an IDE controller, keyboard, southbridge, etc. Again, only the bare minimum initialization is done; Linux does the rest.

When LinuxBIOS is finished initializing hardware, it then looks in the contents of the ROM to find a next stage in the boot process. With the traditional approach of LinuxBIOS, this would be a Linux Kernel. This configuration is what provides such phenomenal records such as 3 seconds from power-on to a bash prompt.

There are however other extensions to LinuxBIOS which make full use of its modular design. A popular choice among these would be to load Etherboot, which gives the bootstrap process the ability to retrieve the next stage in the bootstrap process (either another component to further enhance the boot strap process, or the final operating system itself)[7].

5.2 ADLO

Indeed, instead of executing a Linux Kernel at this stage, we want to load the Bochs BIOS to provide standard commercial PC BIOS interrupt support.

In order to do this effectively, we built a small wrapper program around the Bochs BIOS to transfer valuable information from LinuxBIOS to the Bochs BIOS without having to make modification to the Bochs BIOS. We have named this small wrapper ADLO, the "ADhesive LOader." ADLO allows us to leverage the capabilities of the Bochs BIOS without making modifications to it.

ADLO is responsible for making sure the ROMs that make up the Bochs BIOS and the VGA BIOS are stored at the expected addresses. It also performs the task of copying the Bochs BIOS from its original location into "Shadow RAM." In addition, LinuxBIOS stores some tables (such as a memory map and the IRQ routing table) in a format designed to be practical across architectures, but not conforming to the format in which they're normally stored on a commercial PC BIOS. ADLO converts these tables back to a format easily understood by the Bochs BIOS as it is implemented presently.

In the same vain, the Bochs BIOS was written for the Bochs emulation environment, and was written to emulate an AMI BIOS. To accomplish this goal, configuration of the Bochs BIOS is done by storing and retrieving configuration data in the CMOS as a real BIOS would do. ADLO is responsible for storing some of this data in the CMOS before executing the Bochs BIOS for parameters such as primary boot device and floppy size.

5.3 Bochs BIOS

The primary job of the Bochs BIOS is to set up the Interrupt Vector Table, and supply an entry point for each of its BIOS services. The interrupt vector table is stored from memory location 0000:0000h up to 0000:03FCh and contains a maximum of 256 vectors, each 4 bytes wide.

For instance, if we wanted to save the interrupt vector for our newly implemented Interrupt 13, the vector would be saved as the 19th entry (13h = 19) in the interrupt vector table. Since each interrupt vector is 4 bytes wide, the vector for Int13 will be stored at address 0x4C (19 * 4).

The interrupt handler for Interrupt 13 is placed at offset 0xE3FE within the BIOS image. The BIOS image is placed at segment 0xF000 in memory. Therefore, the four bytes at this offset can merely contain the segment (0xF000) as the high four bytes and the offset (0xE3FE) as the low four bytes.

5.4 Bootloader and Operating System

After the Bochs BIOS has established its interrupt vector table, we rely on a good foundation of BIOS interrupt services to insure that everything continues to run smoothly. To date, we have managed to refine the BIOS services until booting of Linux kernels, which have not been modified for LinuxBIOS compatability, and Windows 2000 have been possible. Once these services had been refined that far, OpenBSD began to boot through to completion as an added reward. Other modern operating systems, for instance FreeBSD and Windows XP, will require further work while still more have gone so far unexplored. All bootloaders we have tried work flaw-

lessly, including the popular Grub and Lilo.

5.5 Hardware Support

To date, the boot strap process outlined in this paper has only been tested on motherboards featuring the SIS630 North/South Bridge chipset. This chipset was chosen as a stable basis for our development efforts due to the long history of support SiS Semiconductors has shown the LinuxBIOS team.

Using any other mainboards in replace of our test system would comprise four steps.

- Insuring that the mainboard is already supported by the LinuxBIOS project[6], or adding support to LinuxBIOS for that mainboard.
- Insuring that LinuxBIOS takes the extra step of initializing the IDE controller. As the Linux Kernel is in most cases capable of this task, it has not already been done for all motherboards with LinuxBIOS supports.
- Extracting of the IRQ routing table for use by ADLO to convey a correct table to the Bochs BIOS.
- Extraction the VGA ROM for use by ADLO if a graphical console is desired.

There are several other hardware issues of interest which are not yet addressed by our efforts.

The first of these is USB support. There are several projects which would benefit greatly from a small open source USB stack. As PS/2 ports have been considered legacy devices for several years now and motherboards are now being sold to consumers which do not feature PS/2 ports at all, the need for simple USB Human Interface Device (HID) support as well as generic USB Storage support has become apparent.

SMP support has not yet been explored. However, LinuxBIOS itself supports SMP on many motherboards, and SMP support has been integrated into Bochs and Bochs BIOS. We expect the integration of the two to be trivial.

6 Conclusions

While open source operating systems on the PC have flourished, the same can not be said for the firmware or BIOS. The reasons for this are many, but the lack of an open source and general purpose BIOS has limited innovation in an important space of the personal computer.

The open source stackable BIOS, described in this paper, eliminates the proprietary BIOS from numerous motherboards, and as a result, a number of new projects can be started within the BIOS space. For example, low level cryptographic support can now be easily added for strong pre-boot authentication, secure remote console support, and secure configuration management are just some of the possible new efforts that can be started now that a general purpose open source BIOS exists.

Source Availability

Instructions for obtaining the source code for this project is located at **http://www.missl.cs.umd.edu/Projects/sebos/main.shtml**

Acknowledgments

Research at the University of Maryland on this project was funded in part by the Composable High Assurance Trusted Systems Program at DARPA via AFRL/IF-NY contract F30602-01-2-0535

Research conducted in the Cluster Research Lab at the Los Alamos National Labs was funded in part by the Mathematical Information and Computer Sciences (MICS) Program of the DOE Office of Science and the Los Alamos Computer Science Institute (ASCI Institutes). 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. Los Alamos, NM 87545 LANL LA-UR-03-2137.

We would like to thank the following people for their hard work, suggestions, and encouragement.

The bochs developers have been an invaluable and helpful group, especially Christopher Bothamy.

We would also like to extend a special thank you to the LinuxBIOS developers, of whom Eric Biederman and Ollie Lho were especially helpful.

The guidance and suggestions of Pascal Dornier and David Sankel made sure that we overcame hurdles

swiftly and easily in bringing all of this work together.

References

[1] Ron Minnich, James Hendricks, Dale Webster. The Linux BIOS. The Fourth Annual Linux Showcase and Conference, October 2000. http://www.acl.lanl.gov/linuxbios/papers/als00/linuxbios.pdf

[2] Phoenix Technologies Ltd. The PhoenixBIOS 4.0 Revision 6 User's Manual. http://www.phoenix.com/resources/userman.pdf

[3] Adam Sulmicki. A Study of BIOS Interrupts as used by Microsoft Windows 2000. http://www.missl.cs.umd.edu/Projects/sebos/winint/index1.html

[4] Adam Sulmicki. A Study of BIOS Interrupts as used by Microsoft Windows XP. http://www.missl.cs.umd.edu/Projects/sebos/winint/index2.html

[5] Bruce M. Simpson. FreeBSD BIOS Coupling on i386. http://www.incunabulum.com/code/projects/freebsd/freebsd-bios-interaction.txt

[6] The LinuxBios Status Guide. http://www.linuxbios.org/status/index.html

[7] The Etherboot Project. http://www.etherboot.org

[8] Ralf Brown's Interrupt List. http://www-2.cs.cmu.edu/ ralf/files.html