FreeVGA: Architecture Independent Video Graphics Initialization for LinuxBIOS*

Li-Ta Lo, Gregory R. Watson, Ronald G. Minnich
Advanced Computing Laboratory
Los Alamos National Laboratory
Los Alamos, NM 87545
{ollie, gwatson, rminnich}@lanl.gov

Abstract

LinuxBIOS is fast becoming a widely accepted alternative to the traditional PC BIOS for cluster computing applications. However, in the process it is gaining attention from developers of Internet appliance, desktop and visualization applications, who also wish to take advantage of the features provided by LinuxBIOS, such as minimizing user interaction, increasing system reliability, and faster boot times. Unlike cluster computing, these applications tend to rely heavily on graphical user interfaces, so it is important that the VGA hardware is correctly initialized early in the boot process in additional to the hardware initialization currently performed by LinuxBIOS. Unfortunately, the open-source nature of LinuxBIOS means that many graphic card vendors are reluctant to expose code relating to the initialization of their hardware in the fear that this might allow competitors access to proprietary chipset information. As a consequence, in many cases the only way to initialize the VGA hardware is to use the vendor provided, proprietary, VGA BIOS. To achieve this it is necessary to provide a compatibility layer that operates between the VGA BIOS and LinuxBIOS in order to simulate the environment that the VGA BIOS assumes is available. In this paper we present our preliminary results on FreeVGA, an x86 emulator based on x86emu that can be used as such a compatibility layer. We will show how we have successfully used FreeVGA to initialize VGA cards from both ATI and Nvidia on a Tyan S2885 platform.

1 Introduction

LinuxBIOS [9] is an open-source replacement for the traditional PC BIOS. The PC BIOS was developed in the 1980's for the original IBM PC, and much of the functionality needed to support this legacy hardware still remains in the PC BIOS today. In addition, the vintage operating systems that ran on these machines were dependent on the BIOS for carrying out many of the configuration activities needed for the system to function properly. Modern operating systems are now able to initialize and configure hardware directly, so there is no longer any reason for the BIOS to be involved. The basic principle behind a modern BIOS, like LinuxBIOS, is to do the minimum necessary to enable system hardware, then leave as much device configuration to the operating system as it can. The result of eliminating this unnecessary initialization is a very fast boot time compared to a traditional BIOS.

Another legacy feature provided by the PC BIOS is a 16-bit callback interface using the x86 software interrupt mechanism. However, only a tiny subset of the interface is used by modern operating systems, and in the case of Linux, it is not used at all. LinuxBIOS does not provide this interface, and as a result, is able to substantially reduce its memory footprint. Compared to 256KB required by the PC BIOS, the typical size of LinuxBIOS is just 32KB to 64KB. This is important because of the small size of FLASH memory in many systems.

Unlike the PC BIOS, only a very small portion of LinuxBIOS is written in assembly code. For the x86 architecture, this is just enough code to initialize the CPU and switch to 32-bit mode. The rest of LinuxBIOS is written in the C language. This makes LinuxBIOS very portable across different architectures, and it has already been ported to support the Alpha and PowerPC processors. Using a high-level language also allows LinuxBIOS to employ a much more sophisticated object oriented device model, similar to the one used in the Linux kernel. In such a model, each physical device has a corresponding software object. The object encapsulates information about the physical device and has methods to probe, initialize and configure the device. The main function of

*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, LA-UR N0. 04-7503

LinuxBIOS is really just to organize, query and manage these device objects. This kind of device object model is unheard of in a BIOS implemented in assembly code.

LinuxBIOS has been successfully deployed in a number of real world applications. At Los Alamos National Laboratory, we have Pink and Lightning [11], two very large production cluster systems that use LinuxBIOS. There are also a number of companies shipping commercial LinuxBIOS-based systems. The advantages of LinuxBIOS have also drawn attention from Internet appliance, desktop, and visualization platform developers. These applications have created a demand for video graphics adapter (VGA) card [7] support under LinuxBIOS. In order to use LinuxBIOS, systems running these applications need to be able to initialize a wide variety of VGA cards. However, LinuxBIOS does not have this capability because it does not provide the 16-bit callback interface mentioned earlier.

Traditionally, a VGA card is initialized by software known as the VGA BIOS, which is considered an extension of system BIOS. It is loaded by the system BIOS from an expansion ROM located on the VGA card into a specific address in system memory. Control is then transfered to the VGA BIOS, and it uses the 16-bit callback interface to communicate with the system BIOS. Since LinuxBIOS does not provide this interface, a non-traditional way to initialize the VGA device in a LinuxBIOS environment is required. In order to achieve this, we have developed a system known as FreeVGA. FreeVGA uses an x86 emulator to run the VGA BIOS. By using an emulator, we free the VGA initialization from any architecture dependencies, since the emulator can operate on any type of processor. In addition, the emulator greatly simplifies the implementation of the 16-bit callback interface.

To demonstrate the effectiveness of this technique, we have used FreeVGA to initialize two VGA cards, an Nvidia FX 5600 and an ATI Radeon 9800 Pro, running on a Tyan S2885 mainboard. Both video cards and the mainboard are state-of-the-art, so we can be confident that FreeVGA will work for virtually all mainboard/video card combinations.

The rest of this paper is organized as follows. Section 2 describes previous work on supporting the initialization of VGA cards with VGA BIOS in non-traditional ways. Section 3 presents how we used an x86 emulator to execute the VGA BIOS and the necessary modification to the emulator and LinuxBIOS itself. We also examine some of the issues that need to be dealt with in order to support this technique. Section 4 shows how we found out the issues described in Section 3 and the strategy we used to overcome these problems. Finally, Section 5 is a roadmap of our future development.

2 Related Work

Initializing VGA cards in a non-traditional way (i.e. not using the standard VGA BIOS in the normal manner) is not a new problem, and various other open source projects have addressed it in the past. There are a number of reasons why VGA cards need to be initialized in this manner.

Due to the limitation of the traditional initialization process and legacy VGA hardware, only one VGA device can be initialized in a given system. For systems with multiple VGA cards, only the first one is initialized at boot time, other cards have to be *soft-booted* after the operating system is loaded.

Some VGA hardware is fragile, so that a slight error in programming the registers will put the hardware in a non-functional state. A complete re-initialization is then required to bring the hardware back to normal. Normally, the only way to do such a re-initialization is to re-execute the initialization code in the VGA BIOS. Of course it is always possible to reset the whole system as a last resort, but usually this is not an option. In these cases it must be possible to reset just the VGA device while other parts of the system are still running.

Other open source projects that have addressed the need to initialize VGA cards are described in the following sections.

2.1 SVGALib

SVGALib [2] is a library that provides a generic VGA interface for older VGA cards. It includes a utility called `vga_reset` to re-initialize VGA cards. The utility uses the vm86 mode of x86 processors to execute the VGA BIOS. In vm86 mode, an executing program is just like any other program executing in 32-bit mode, but it executes 16-bit code like a traditional 8086 CPU. To support this, Linux provides a system call to switch a process into vm86 mode. `vga_reset` first maps in the BIOS code and data area from physical memory space to its virtual memory space. Then it sets up register values for instruction and stack pointers. Finally, it enters vm86 mode by calling the vm86 system call.

By default, both VGA BIOS and system BIOS callbacks are executed natively by the hardware, except some privileged instructions and I/O operations. By giving different flags when entering the vm86 mode, it is possible to choose to intercept I/O and BIOS calls. This feature was used frequently in the early stage of the development of our solution as a debugging and verification tool. The I/O and BIOS call logs from `vga_reset` and x86emu were compared to examine if both `vga_reset` and x86emu had the same code execution path in the same hardware environment. If they both had the same

execution path, it indicated that the emulator executes the VGA BIOS exactly as the real hardware.

The disadvantage of `vga_reset` is that the vm86 mode is not supported by the AMD x86_64 architecture. The 64-bit Linux kernel does not provide the vm86 system call. During our development, we had to install a 32-bit Linux distribution on our 64-bit AMD K8 platform to run `vga_reset`.

2.2 ADLO

ADLO [5] was the original effort to add VGA BIOS support into LinuxBIOS. ADLO uses the BOCH BIOS to replace the traditional system BIOS. It loads the BOCH BIOS and VGA BIOS into the memory addresses where the traditional system and VGA BIOS are loaded. ADLO then switches the processor back to 16-bit mode and jumps to the entry point of the BOCH BIOS. The BOCH BIOS tries to do the same initialization process and to provide the same BIOS callbacks as a traditional BIOS. The net effect of this BOCH+VGA BIOS combination is like a software reboot in a traditional BIOS system. One of the advantages of ALDO is that it can support legacy operating systems like Window 2000. The problem of ADLO is that it depends on the BOCH BIOS, which is very difficult to maintain and modify. Even adding a message printing statement in the BOCH BIOS will make it fail to build.

2.3 VIA/EPIA Port

Another effort to use VGA BIOS to initialize VGA hardware is the VIA/EPIA port of LinuxBIOS. The port uses a *trampoline* to switch back and forth between the 16-bit and 32-bit modes of x86 processors. This allows the VGA BIOS to be executed directly in 16-bit mode but standard BIOS callbacks to be emulated in 32-bit mode in the C language. Before executing the VGA BIOS, a 16-bit interrupt descriptor table (IDT) is set up to redirect all interrupt calls to the trampoline. The BIOS then switches to 16-bit mode and jumps to the entry point of VGA BIOS. When the VGA BIOS calls the standard BIOS callbacks, the trampoline switches to 32-bit mode, and dispatches the call to the BIOS emulation code. After the emulation code returns, the trampoline switches back to 16-bit mode and returns to VGA BIOS. The main limitation with this approach is that it is highly architecture specific, so can't be used for non-x86 based architectures. The other disadvantage of this method is that because the trampoline is inside LinuxBIOS, it is more difficult to debug than a user space program like x86emu.

2.4 XFree86

The XFree86 project [4] has to solve this problem in order to support multiple-card, multiple-screen configurations. Since XFree86 supports multiple architectures, the solution must be able to initialize VGA hardware not only on x86 systems, but also on other architectures like Alpha and PowerPC. To achieve this, XFree86 uses x86emu emulator to execute the VGA BIOS directly. X86emu is an x86 instruction emulator which does not emulate any hardware other than the core x86 processor in 16-bit mode. The emulator provides helper function stubs to access I/O and memory spaces. XFree86 implements these helper functions in architecture dependent ways. XFree86 first *unmaps* the primary VGA card from physical I/O and memory spaces by programming well known legacy I/O ports or registers in the PCI configuration space. Then it maps in the I/O and memory spaces of the secondary card, loads the VGA BIOS image of the card into the *virtual* memory space of the emulator. The emulator then executes the VGA BIOS starting from the entry point. Because most VGA BIOSes also require traditional BIOS callbacks which are not available in non-x86 systems and not usable in 32-bit x86 systems, the emulator also has to intercept these BIOS calls and then emulate them in a similar way as I/O and memory accesses.

In a summary, each of these previous efforts was found to have deficiencies. Both SVGALib and VIA/EPIA are non-portable, so are not suitable for integration in LinuxBIOS. SVGALib, ADLO and VIA/EPIA were found to be very difficult to debug, which is a major problem for a complex system like LinuxBIOS. XFree86 initializes the VGA hardware very late, which does not meet our design goals. These problems motivated us to seek a solution which was able to address all these issues, while taking advantage of the experience gained by the previous research.

The concept and advantages of using an emulator to execute the VGA BIOS in order to initialize VGA hardware are obvious. The solution is portable across different platforms, and it is flexible enough to monitor I/O and memory accesses and BIOS callbacks. The ability to monitor these accesses is very useful for debugging purposes. As a consequence, the solution we have chosen for FreeVGA is based on a modified version of x86emu.

3 VGA Emulation

Most modern VGA cards support two modes of operation: legacy mode and native mode. In legacy mode the card replicates the graphics hardware interface that was used on the original IBM PC/AT. The legacy mode

is primarily used to provide a compatibility mode so that applications and drivers have a common programming interface. In native mode, the card provides access to a vendor specific register interface that is used to configure and control the card. Most VGA cards require an elaborate programming sequence to initialize the VGA hardware and turn it to legacy mode. If LinuxBIOS was to support direct initialization of the card, it would need to use a device driver that provided this programming sequence. Unfortunately, most vendors worry that even exposing the interface to their proprietary hardware will allow their competitors to plunder their intellectual property; hence they do not reveal the sequence of register diddles necessary to initialize their cards.

Linux uses a frame buffer device as an abstraction of graphics hardware. Applications interact with frame buffer device interface (`/dev/fb`) instead of directly accessing the hardware. At a minimum, the frame buffer device driver provides support to switch video modes and some basic drawing functionality. Some vendors like Matrox provide a *sophisticated* frame buffer device driver which can initialize the hardware from the power-up state to any video mode available by the hardware. Our original intention was to persuade vendors to provide these sophisticated drivers so that they could be used by LinuxBIOS. However many of the vendors who provide enough information to implement a minimal frame buffer device driver hesitate to provide the additional information necessary for a such a driver.

As a result, the only way to reliably initialize the hardware from power-up in a vendor-neutral manner is to run the vendor supplied VGA BIOS. Once the VGA BIOS has been run, the card will switch to legacy mode and it can be controlled using the legacy interface from then on. Depending on the implementation of the VGA BIOS, the 16-bit BIOS callback interface may be used to communicate with the system BIOS. Since LinuxBIOS lacks this callback interface, it can not support VGA BIOS directly in the same way as the traditional system BIOS. We have to either add the 16-bit callback interface to LinuxBIOS or use another software to provide this interface. The use of an emulator solves this problem by allowing the emulator to run in 32-bit mode to execute the 16-bit mode VGA BIOS and then implement the callback interfaces as necessary.

3.1 x86emu

The emulator we used to enable VGA support in LinuxBIOS was based on a modified version of x86emu. The x86emu emulator was originally developed by SciTech Software [1] as part of their SciTech SNAP SDK. The XFree86 project adopted the emulator for soft-booting VGA cards in their X-server. We used the XFree86's version rather than the SciTech's version because it is more updated and debugged.

The virtual machine of the emulator is implemented by a data structure representing all the integer and floating point registers in an x86 CPU. The emulator decodes and jumps to an entry of a function table based on the first op code of each instruction. The functions in the function table update the virtual machine with the outcome of the *execution* of the instruction. The emulator uses helper functions provided by client applications to communicate to the real world, for instance, accessing I/O and memory spaces. The emulator allows interrupt handling using either an interrupt handler provided by the client application or an interrupt handler in the BIOS.

IO and Memory Access The client application provides a set of functions for accessing I/O ports and another set of functions for accessing memory addresses. The structures used to define the functions are shown below:

```
typedef struct {
    u8    (inb)(int addr);
    u16   (inw)(int addr);
    u32   (inl)(int addr);
    void  (outb)(int addr, u8 val);
    void  (outw)(int addr, u16 val);
    void  (outl)(int addr, u32 val);
} X86EMU_pioFuncs;

typedef struct {
    u8    (rdb)(u32 addr);
    u16   (rdw)(u32 addr);
    u32   (rdl)(u32 addr);
    void  (wrb)(u32 addr, u8 val);
    void  (wrw)(u32 addr, u16 val);
    void  (wrl)(u32 addr, u32 val);
} X86EMU_memFuncs;
```

These two sets of functions are installed into the emulator via `X86EMU_setupPioFuncs()` and `X86EMU_setupMemFuncs()` respectively.

Since we are working on an x86 platform, the implementation of the I/O access functions is just a thin wrapper for the inline assembly functions provided in `sys/io.h`. All I/O operations are directed to the physical I/O ports without any intervention or emulation.

In our setup, we statically allocated 1MB of memory to be used as the virtual memory of the emulator. The memory access functions direct all memory accesses made by VGA BIOS to this area, except accesses to the legacy VGA buffer. These are directed to another virtual memory area which is mmaped from `/dev/mem`. We implemented the memory access functions by reading and writing these two memory regions according to an address passed as argument to these access functions.

Interrupt Handling In the x86 architecture, there are 256 software interrupts. The client application provides an array of 256 functions to the emulator for interrupt handling, in a similar way as I/O and memory access functions. When the emulator encounters an `INT` instruction with an interrupt number `N`, it calls the `N`th entry of the array. The interrupt handling function can choose to handle the interrupt by itself or let the emulator execute the handler in its virtual memory.

In our implementation, all software interrupts are directed to a single `do_int()` function. When this function is called with a interrupt number, it first checks if there is any handler installed by the VGA BIOS for that interrupt number. If there is no handler installed, it will call the default emulation code implemented in the C language, otherwise it will execute the handler installed by VGA BIOS with the emulator.

3.2 Legacy VGA Issues

Using x86emu provides LinuxBIOS with a means of initializing the VGA hardware and then switching the card to legacy mode. However there are a number of other issues that need to be addressed when the card is operating in this mode. Figure 1 shows the system and card memory layout when operating in legacy mode.

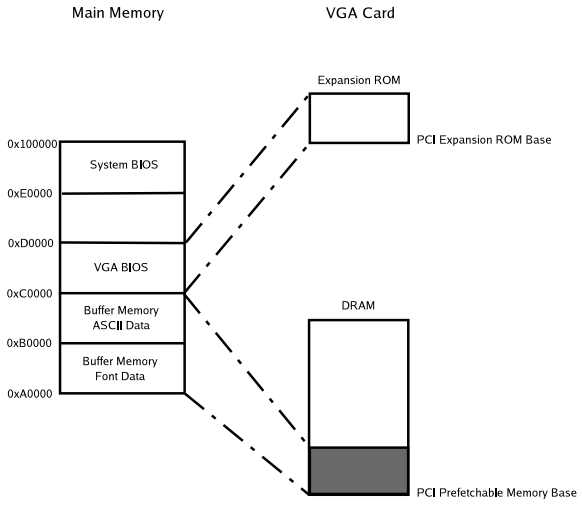

Figure 1: Legacy VGA Memory Map

Buffer Memory The memory on the VGA card is mapped to system physical address space by the prefetchable memory resource in the PCI Configuration Space of the card. The buffer memory used in legacy mode is just a small portion of the whole memory installed on the card, as shown in the shaded area in Figure 1. This portion of memory is mapped to system memory addresses 0xA0000 to 0xBFFFF respectively. Part of this area is used to store bitmap data which is interpreted by the hardware as font information. The rest is used to store the ASCII codes and color values to be displayed on the screen. The VGA BIOS clears and updates the buffer memory during initialization, so it is necessary to map this region of physical memory to the virtual memory space of the emulator.

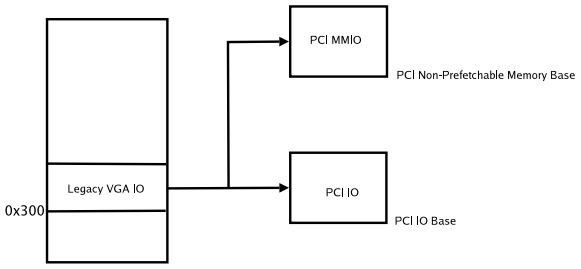

Figure 2: Legacy VGA I/O Map

I/O Addresses The control registers of the VGA device are located by the PCI I/O resource map to the I/O space of the processor, or by the PCI non-prefetchable memory (MMIO) resource map to a range of physical memory addresses. VGA cards also provide access to control registers via the legacy VGA I/O addresses in the range 0x300 to 0x400. Generally, the VGA BIOS will access control registers using both memory mapped I/O and via the legacy VGA I/O ports. To support this, it is necessary to ensure that both I/O access methods are forwarded correctly to the VGA device.

Expansion ROM Before the VGA BIOS can be executed, it has to be loaded from the expansion ROM on the VGA card into the VGA BIOS memory area in system memory. The PCI specification [10] defines the format for the VGA BIOS image and a procedure to load the image as follows:

1. The image in the expansion ROM starts with a 0x55, 0xAA signature.

2. The system BIOS should search the for signature in the expansion ROM and load the image into memory.

3. If the device is a VGA device, the system BIOS is required to load the image to 0xC0000 which is the VGA BIOS area.

4. After loading the image, the system BIOS should jump to the entry point of the image which is at offset 0x3.

The expansion ROM loading was implemented in LinuxBIOS as the initialization methods of PCI devices. LinuxBIOS probes and allocates the expansion ROM resources required by PCI devices in one of the stages of device enumeration. In a later stage, it loads the expansion ROM image from ROM to RAM and uses the emulator to execute the image.

CPU Cache The system memory region allocated to VGA buffer memory (0xA0000-0xBFFFF) is actually aliased in legacy mode. This means that memory accesses in this range can be forwarded to system memory or VGA card memory depending on the setting of the system chipset or the cache controller in the processor.

The cache in x86 processors after Pentium Pro is configured by Model Specific Register (MSR) called Memory Type Range Register (MTRR) in the processor. MTRR controls the cache mechanism of a range of physical address. Addresses under 1MB are controlled by *fixed* MTRRs and addresses above 1MB are controlled by *variable* MTRRs. The fixed MTRRs on the K8 have 2 extension bits (RdMEM and WrMEM) [6] which control the forwarding of read and write access to memory address under 1MB. When set and enabled, the RdMEM bit in the MTRR will forward read access in the range to system memory. The WrMEM bit will forward write access to system memory. However, since we want memory access to the VGA buffer memory be forwarded to the VGA card, we have to clear these two bits in the MTRRs controlling this memory range.

HyperTransport Routing The AMD K8 processor and its chipsets are interconnected by HyperTransport Technology [8]. Each processor has a *northbridge* integrated in the same package. The northbridge connects the core processor to system memory and other parts of the system (via a *southbridge*). On the K8, each northbridge provides three HyperTransport links that can be used for communication. The way the processors and chipsets are connected via these links is determined by mainboard designers and varies from board to board. There is also a HyperTransport routing table in the northbridge which controls how memory and I/O requests are routed. The BIOS has to configure the routing table based on both the physical layout of the HyperTransport hierarchy, and the PCI devices attached to the hierarchy. I/O and memory transactions can then be forwarded to the correct HyperTransport link on the northbridge. For VGA devices, accesses to both the I/O and memory resources defined in the PCI configuration space, and to the legacy VGA, also have to be forwarded correctly.

Figure 3 shows the HyperTransport hierarchy of the Tyan S2885. The mainboard designers have connected the two CPUs together by *Link 1* on each CPU. The

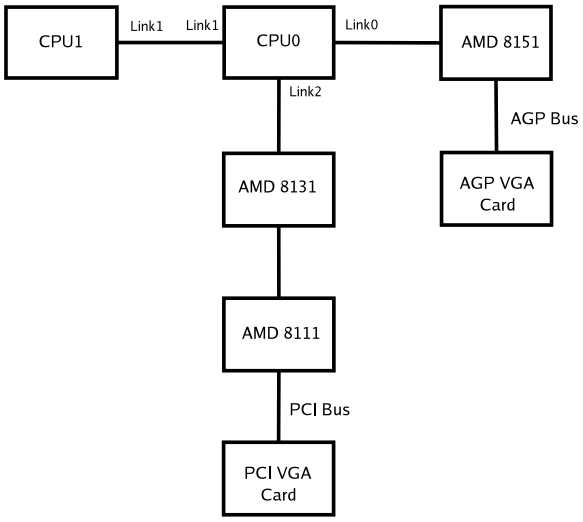

Figure 3: S2885 HyperTransport Hierarchy

AMD 8151 AGP bridge is connected to *Link 0* and the AMD 8131 PCI-X bridge is connected to *Link 2* on CPU 0. *Link 0* and *Link 2* of CPU 1 are left unconnected. LinuxBIOS must configure the routing table based on which kind of VGA card is installed. For example, for a VGA card connected to the AGP, it is necessary to set up the routing table in CPU 0 to forward legacy VGA I/O and memory transactions to *Link 0*. However if the VGA card is connected to the PCI bus, the routing table will need to forward transactions to *Link 2*.

PCI and AGP Bridges PCI and AGP bridges forward I/O and memory transaction from its primary bus to its secondary bus. The range of access to be forwarded is configurable and should cover the PCI I/O and memory resources used by all devices on the secondary bus. Usually this range does not include the I/O amd memory addresses used by legacy VGA. We have to enable forwarding of legacy VGA access in additional to normal PCI access by programming a bit in the Configuration Space of the bridge.

3.3 Integration

In order to achieve our objective of early VGA initialization during the BIOS boot phase, it is necessary to make the emulator part of LinuxBIOS.

User Space Versus Kernel Space One problem we expected to encounter when moving the emulator into LinuxBIOS was operating system and library dependency

issues. The emulator was a user space program that used operating system and library calls for memory management, message printing and accessing PCI configuration space. Those functions are not available during the boot phase, and need to be replaced by corresponding supporting routines in LinuxBIOS.

Fortunately, it turned out that the integration process was relatively easy, since there were only a few operating system and standard library dependencies in the emulator itself. Most of the effort was spent on implementing expansion ROM loading in LinuxBIOS, and fixing bugs in I/O and memory transaction forwarding.

Device Object Model The integration fully used the device object model available in LinuxBIOS. This was done in such a way that not only VGA BIOS could be executed, but the same technology could also be used to initialize other devices requiring a proprietary BIOS, for instance, some SCSI controllers.

The emulator was treated as one of the initialization methods of PCI devices. This fitted the emulator nicely into the LinuxBIOS device driver model because the initialization method is automatically called at certain stages of the device enumeration. Once the device enumeration code found a PCI device with expansion ROM, it would call the emulator at the appropriate stage to initialize the hardware. The hardware would then be initialized in the same fashion as any other devices are initialized.

With this technique, the VGA hardware is fully configured before any bootloader payloads are loaded. This means that bootloaders can now use the legacy mode of the VGA device as a console and the Linux console driver works in exactly the same way as in a traditional PC BIOS environment.

Size Overhead Integrating FreeVGA into LinuxBIOS had virtually no impact on the size of the resulting ROM image. The compressed ROM image only increased by 16KB, but because the final ROM image is padded to the nearest power of 2, this increase was absorbed into the existing unused space. The runtime size of the uncompressed image was only increased by 40KB.

4 Testing

Testing of FreeVGA was carried out on a Tyan S2885 mainboard. This mainboard was chosen because it was a state-of-the-art board that uses 64 bit AMD CPUs. It was also the only AMD K8 mainboard with an AMD 8151 AGP bridge and an AGP slot. The AGP slot on the mainboard allowed us to test a range of newer generation AGP VGA cards. The mainboard was configured with dual 1.6 GHz AMD K8 processors. There was 3GB of DRAM installed, 2 GB of the memory was installed on DRAM DIMM connected to CPU 0 and the other 1GB was connected to CPU 1.

One of the challenges of the Tyan was that it was a very complex platform to work with. The first time we tried running the emulator we received nothing at all on the screen. Checking the execution log from the emulator, we found that I/O accesses to the PCI I/O resource region of the VGA device returned meaningful values, but that I/O accesses to the legacy VGA I/O ports on the card always returned invalid values. From this we were able to deduce that the northbridge and AGP bridge were forwarding normal PCI I/O accesses to the card correctly, but accesses to the legacy VGA were not. To test this, we temporarily configured legacy VGA forwarding in the AGP bridge and HyperTransport routing in the northbridge. Re-running FreeVGA at this point resulted in scrambled text on the screen. This was promising, and it verified that HyperTransport routing was going to be crucial for this platform.

Next we tried to alter the scrambled pattern by writing to the buffer memory via both the legacy VGA buffer area and the PCI memory resource region. We found that changes made via the PCI memory resource region were not reflected in the legacy VGA buffer area and vice versa. This was strange because in theory no matter which way the buffer memory was modified, it should updated the same memory on the VGA card. We finally realized that the cache controller in the AMD K8 processor was forwarding the access to the memory on the mainboard rather than on the VGA card. This problem was solved by changing the MTRRs with the help of the kernel MSR driver.

At this point, the screen remained scrambled even though we were sure that the CPU was forwarding accesses correctly. By comparing the contents of the VGA buffer memory when the system was booted with both a tradition BIOS and with LinuxBIOS, we found that the contents of the font data memory were different. This was because the emulator was executing the VGA BIOS in its own virtual memory address, whereas the VGA BIOS tried to update the font data at a physical address. This was solved by modifying the emulator to map the physical memory device `/dev/mem` to its virtual memory address for the legacy VGA buffer.

4.1 Nvidia FX 5600

The first VGA card we tried was an Nvidia FX 5600. This is a high performance 3D graphics card with 256MB of onboard memory.

In additional to the fact that Nvidia is the market leader in VGA chipset design, we choose this card be-

cause Nvidia have never publicly released any programming documentation on the hardware. This meant that FreeVGA would not be able to do any direct card configuration and would have to rely totally on running the vendor supplied VGA BIOS. Much to our surprise, executing the VGA BIOS work successfully on the first attempt and we were greeted with the Nvidia banner messages on the screen. It turned out that the VGA BIOS did not do anything unusual, nor did it require any BIOS callbacks. After the successful initialization, we were able to run Linux VGA console without problem.

In order to fully exercise the Nvidia card, we ran a benchmark using the Unreal game [3] under the XFree86 X-window system. This also allowed us to test two different versions of the X server driver, one provided by the XFree86 Project and the one provided by Nvidia. Both drivers worked flawlessly and there was no significant difference running the benchmark as compared to the system booted with a traditional BIOS. This benchmark also exercised the 3D and AGP operation of the card and no problems were found.

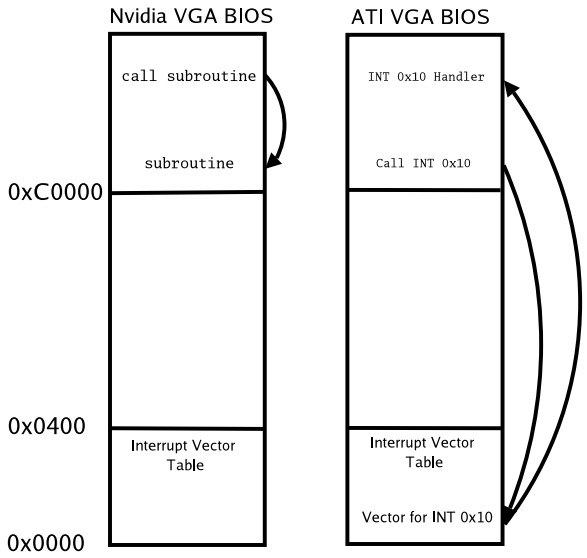

Figure 4: Direct and Indirect BIOS call

4.2 ATI Radeon 9800 Pro

The ATI Redeon 9800 Pro is a high performance 3D graphics accelerator card with 256MB of memory. The card was manufactured and supplied by Tyan. It is comparable in performance to the Nvidia card.

The first time we tested the emulator on the ATI card it crashed the system. By examining the I/O logs we found that during the initialization, the VGA BIOS was setting the video mode and then displaying some messages on the screen. As shown in Figure 4, the Nvidia BIOS did this by calling subroutines in the BIOS directly. In the case of the ATI card, the VGA BIOS tried to install a BIOS callback handler in the interrupt vector table and then call the handler using the software interrupt mechanism. Although using a BIOS callback was not a problem for FreeVGA, the problem was that we were intercepting the software interrupt and implementing our own handler base on our limited knowledge of legacy VGA. Obviously, the way we implemented the handler was incorrect for the ATI hardware and caused the crash. Once we stopped intercepting the BIOS call and executed the handler in the ATI BIOS, the card was initialized without problem.

At this point we ran the same benchmark on the ATI card. This verified that there were no differences in the performance and operation of the card compared to a system booted with the traditional BIOS.

5 Future Work

The work to date has shown that it is possible to use our methodology to reliably initialize two very different VGA cards. Because of the different nature of the cards, and the fact that we have not needed any vendor input to achieve this result, we are confident that this technique will apply to virtually any type of VGA card. However there are still a number of issues that need to be addressed before FreeVGA is ready for general use.

Chipset Dependencies At the time of writing, we have only tested the emulator on an AMD K8 platform. Each vendor chipset has its own, very different, way of caching the frame buffer memory and forwarding I/O and memory accesses. The HyperTransport architecture of the AMD K8 is the most complicated one we have ever seen so far, however there is no guarantee that the techniques we have used here will be applicable to other chipsets. More testing is required so that that the experience we have gained on AMD K8 can be extended to the support of other chipsets.

Other Architectures One of the advantage of FreeVGA is its architecture independence. Since LinuxBIOS already supports other non-x86 architectures, such as the PowerPC, it will be necessary to port FreeVGA support these other architectures too. Currently, VGA cards have to be programmed using OpenFirmware instead of the x86 VGA BIOS in the expansion ROM to be usable on PowerPC. This has severely limited the choice of VGA cards on PowerPC-based system. By using FreeVGA to initialize VGA cards on PowerPC, the same VGA cards that are

available for the x86 architecture will be available for the PowerPC.

The main issue of porting the emulator to these architectures is that they have very different ways of accessing legacy VGA IO and memory. The legacy VGA IO and memory are mapped to physical memory address in a chipset and mainboard dependent way. The mechanism of accessing PCI Configuration Space is different from x86 architecture too. We expect a much more complicated implementation of `X86EMU_pioFuncs` and `X86EMU_memFuncs` for these architectures.

6 Conclusion

In this paper, we have described FreeVGA, an architecture independent method for initializing video graphics adapter cards. The technique was developed so that LinuxBIOS, an open source replacement for the traditional PC BIOS, would be able to initialize graphics hardware very early in the boot process. To achieve this, FreeVGA uses an x86 emulator based on x86emu to run the actual VGA BIOS from the graphics card. This ensures that the card is initialized correctly, and does not require any knowledge of proprietary hardware information.

FreeVGA has been successfully tested using a Tyan S2885 mainboard configured with both ATI Radoen 9800 and Nvidia FX 5600 cards. Our testing showed that these cards could be successfully initialized with FreeVGA, and then support the operation of the XFree86 X-window system without any problems. Both the AGP and 3D features of the cards were completely operational, and benchmarking showed no performance difference compared to the system booted with the standard PC BIOS. Although there are still a number of issues to be addressed to enable seamless integration with LinuxBIOS, the results of our testing gives us great confidence that FreeVGA will be an effective, vendor independent, alternative for initializing VGA hardware on a range of different platforms.

7 Acknowledgment

The author would like to thank David Hedricks for setting up the development platform and performing the elaborate testing.

Tyan Computer Corp. kindly provided the S2885 mainboard and the ATI Radeon 9800 Pro card. Tyan is a long time supporter of the LinuxBIOS project, and ships LinuxBIOS on a number of its mainboard products.

References

[1] http://www.scitechsoft.com/.

[2] http://www.svgalib.org/.

[3] http://www.unrealtournament.com/.

[4] http://www.xfree86.org.

[5] Adam Agnew, Adam Sulmicki, Ronald Minnich, and Willian Arbaugh. Flexibility in rom: A stackable open source bios. In *2003 USENIX Annual Techinical Conference*, San Antonio, Texas, USA, June 2003.

[6] AMD. *BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors*, May 2003.

[7] Richard F. Ferraro. *Programmer's Guide to the EGA, VGA, and Super VGA Cards*. Addison Wesley, 1994.

[8] HyperTransport Technology Consortium. *HyperTransport I/O Link Specification*, January 2003.

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

[10] PCI-SIG. *PCI Local Bus Specification*, December 1998.

[11] Gregory R. Watson, Matthew J. Sottile, Ronald G. Minnich, Erik A. Hendriks, and Sung-Eun Choi. Pink: A 1024-node single-system image linux cluster. In *Proceedings of HPC Asia 2004*, Toyko, Japan, July 2004.