Linux* in a Brave New Firmware Environment

Matthew Tolentino
Intel Corporation
Enterprise Products & Services Division
`matthew.e.tolentino@intel.com`

Abstract

Initially included exclusively on Intel®[1] Itanium®[2] platforms, the Extensible Firmware Interface Architecture (EFI) will soon be supported on IA-32 server, workstation, and desktop systems. This paper provides insight into the design and composition of an EFI enabled IA-32 Linux kernel capable of booting on legacy free platforms. An overview of the EFI development environment is provided, including the specifications, development tools, and software development kits available for development today.

The design and prototype implementation of the kernel initialization sequence from the instantiation of the EFI enabled Linux boot loader to the login prompt is detailed with an emphasis on maintaining backward compatibility with existing legacy platforms. The legacy free VGA replacement, the Universal Graphics Adapter (UGA), is introduced in the context of Linux, including the requirements for use within the kernel. Additionally, details of a prototype implementation of the Universal Graphics Adapter Driver stack, including an EFI Byte Code Interpreter and Virtual Machine (EBC VM), are presented and analyzed. This paper concludes with a call for kernel developers to review and provide feedback on the design and implementation presented.

1 Introduction

This paper begins with an overview of the Extensible Firmware Interface and the Universal Graphics Adapter. The architecture of an EFI enabled Linux kernel is presented as well as the design and implementation details of the EFI Linux Boot Loader, kernel initialization changes, and support for the Universal Graphics Adapter. Future enabling work is outlined and conclusions presented.

2 EFI Overview

The EFI Specification defines a consistent, architecturally neutral interface between platform firmware and operating systems. Designed to address the limitations and issues inherent in legacy BIOS support for PC-AT systems, EFI provides a core set of services and protocol interfaces used to initialize platform hardware as well as common interfaces to access platform capabilities. Additionally, EFI aggregates platform configuration information that operating systems require for initialization – information that has been traditionally obtained through BIOS calls. This includes details ranging from the system memory map and the number of processors in the system to the

*Linux is a trademark of Linus Torvalds

[1]Intel is a registered trademark of the Intel Corporation

[2]Itanium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries

parameters of the current video console. Further, the structure of EFI is modular such that as incarnations of new technologies are incorporated into systems, the interfaces to the operating system for these technologies will not require significant modifications. The system level view of the conceptual design of EFI is depicted in figure 1.

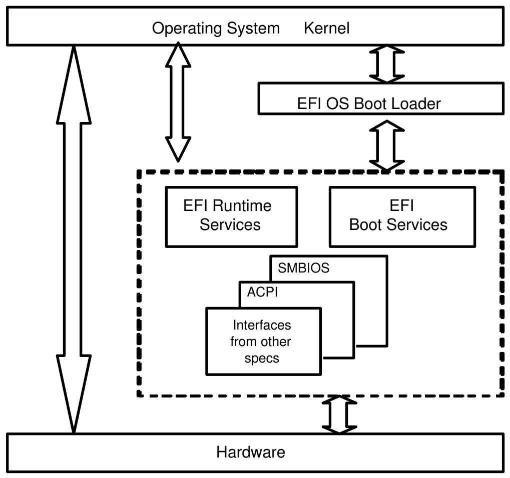

Figure 1: EFI System View

Once the system is powered on, the system firmware initializes and owns all hardware resources. Before an operating system is loaded, EFI provides a pre-boot environment and manages platform resources through the Boot Services and Runtime Services tables. These tables, encapsulated in the EFI System Table provide access to system resources. Unlike legacy BIOS which executes in 16bit real mode, EFI provides a 32bit protected mode operating environment.

2.1 EFI Boot Services

Accessed through the EFI System Table, EFI Boot Services provide platform independent functionality that is only available before an operating system is loaded. This includes services such as memory allocation routines, device access protocols, time services, and others detailed in [1]. Once control of the system is transferred to the operating system, EFI Boot Services are terminated. The OS loader is required to call ExitBootServices() as part of the transition of control to the operating system.

2.2 EFI Runtime Services and Drivers

The EFI Runtime Services provide OS neutral platform specific functionality that persists into OS runtime. One example of an EFI Runtime driver is the floating point software assist driver (fpswa.efi) on Itanium platforms discussed in [4]. Another example supported by both IA-32 and Itanium platforms is the Universal Graphics Adapter driver. EFI implementations may provide any number of EFI drivers for use during OS runtime in order to take advantage of the generic functionality. Details of these drivers are passed to the operating system via the EFI Configuration Table and the memory locations occupied are specified as Runtime Memory.

2.3 EFI Configuration Table

Leveraging existing standards and technologies, the EFI configuration table provides an interface to commonly used tables that describe platform resources. Each entry in the Configuration Table is comprised of 2 elements - a Globally Unique Identifier as defined in [9] and a pointer to the respective table. Examples of configuration table entries include ACPI tables, UGAs discovered by the firmware, and SMBIOS tables.

2.4 EFI Boot Loader

Generally, loading and initializing an operating system involves several steps. The first step is the determination of the kernel image to load into memory as well as any additional required components, such as a RAM disk, that are required. The second is the act of loading the chosen images into memory. The third and final step involves transferring control to the loaded kernel, thus enabling the kernel initialization sequence to commence.

Several boot loaders have been developed to load Linux on IA-32 platforms, including the popular lilo and grub loaders described in [7] and [8] respectively. Each of these provides the capability to boot Linux kernels from a legacy BIOS and offer varying degrees of functionality as discussed in [3]. However, these loaders do not provide the capability to boot from the EFI environment. In order to boot a legacy-free Linux kernel from EFI, an EFI native application is necessary to set up and affect the transfer of control from the firmware to the kernel.

3 UGA Overview

The UGA is a software abstraction that provides an interface for simple graphical output that does not require specific implementation knowledge of the video hardware. UGA serves as a replacement for VGA hardware and video BIOS providing graphical display capabilities in an image that can be used by both system firmware and operating systems. Unlike VGA, UGA enables several significant features including support for controlling multiple output devices and higher default screen resolutions. UGA ROMs may reside anywhere in memory, alleviating the need for static reservation of specific video RAM and ROM memory regions. Additionally, multiple UGA ROMs may be used in a single system without conflict.

A key design consideration of UGA is the processor and platform independent nature of the driver image. Compilation of driver images into EFI Byte Code (EBC) enables a single image to execute on multiple architectures. The resultant byte code image must be interpreted and executed in the context of an EBC Virtual Machine capable of translating EBC instructions to native instructions.

Because the UGA is an EBC image and operating systems are expected to use the same image as the pre-boot firmware environment, an OS resident EBC Virtual Machine is required. This Virtual Machine provides an EFI execution environment within the OS, interfaces with the console layer and the PCI subsystem of the operating system, and provides the means to interpret and execute the EBC instruction stream of the UGA. Figure 2 pictorially describes this generic design.

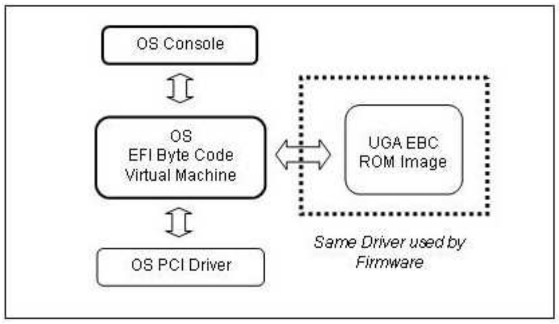

Figure 2: UGA Conceptual Design

Although UGA does provide graphics display capabilities, it is not intended to replace high-performance, operating system specific graphics drivers. Rather, the UGA is specified for use in the absence of a performance oriented graphics driver. For example, a back-end server may not require extensive graphics-oriented functionality during normal operation. There may also be cases where an installation kernel may require the user to provide a graphics driver. In these cases, a UGA driver may be used as a default console driver. Further advantages of UGA can be found in section 10 of [1].

3.1 OS Console Driver

Support for UGA at OS runtime requires an interface to the OS specific console subsystem. This driver is responsible for affecting change to the video controller via the UGA protocols outline in section 10 of [1].

3.2 OS EFI Byte Code Virtual Machine

The OS EBC Virtual Machine, as depicted in figure 2, serves two functions. The first is to provide an EFI emulation environment functionally equivalent to the EFI pre-boot environment. This enables the use of the same UGA driver as firmware through emulation of the EFI Boot Services.

The second function is to provide the facility to decode and execute EBC instructions. UGA drivers compiled into EBC can not be directly executed. Therefore, an OS present mechanism is needed translate the EBC instruction stream of the UGA image into instructions of the native processor.

3.3 OS PCI Driver Interface

In pre-boot space, the UGA driver uses the PCI I/O Protocol, a Boot Services structure, to access the video controller. Because this mechanism is no longer available when the operating system takes control, PCI access must be provided via an OS level implementation of the PCI I/O Protocol interface structure. This enables the UGA to use the standard operating system interface to access PCI space.

3.4 UGA EBC ROM

The OS support for UGA is capable of executing any UGA EBC ROM discovered by firmware that is compliant with the EFI Driver Model. In order to facilitate the transfer of control from EFI to the operating system, UGA drivers used in pre-boot space will be halted upon termination of Boot Services. The image must be re-initialization to be used during OS runtime.

4 Architecture of an EFI enabled Linux Kernel

The advent of EFI on legacy free, IA-32 systems necessitates additional support in several key areas of the Linux kernel. This section presents an overview of the architectural differences in booting the kernel from EFI versus legacy BIOS as well as the impact of the changes. The details of the changes specific to these areas are discussed in the remainder of this paper.

4.1 Key Differences

One of the key differences in booting from EFI versus legacy BIOS on IA-32 systems is the capability to launch the kernel in 32bit, protected mode. The firmware no longer invokes the boot loader in 16bit real mode. As a result, the kernel no longer needs to affect the transition to 32bit, protected mode.

Loading an EFI enabled kernel requires an EFI Linux boot loader. The loader is a native EFI application, which is responsible for obtaining platform configuration information EFI and loading the kernel into memory. Included as boot parameters, all platform configuration information is collected by the loader and passed to the kernel. This permits the loader to transfer control directly to the architecturally specific entry point of the kekernel. This difference alleviates the need for the kernel code that collects sytem information via BIOS calls.

On EFI platforms, EFI defined services are employed to invoke firmware functionality as opposed to legacy BIOS calls. The EFI Runtime Services provide a standard interface for accessing firmware and hardware in a platform independent manner. For example, access to the real time clock is accessed via a firmware function in the Runtime Service table as opposed to directly reading CMOS. The practice of scanning memory to find the signatures of hardware description tables is no longer necessary because tables such as ACPI are included as part of the EFI interface.

The advent of the Universal Graphics Adapter no longer requires static reservation of fixed memory regions and presents the opportunity for kernel level multi-head console support.

The design of an EFI enabled kernel requires key modifications to the following three crucial areas:

- Boot loader
- Kernel initialization sequence
- Console subsystem

4.2 Impact of EFI Kernel Changes

The changes to the kernel to support booting from EFI are relatively straightforward. The modifications to the boot loader involve extending the functionality of the IA-32 aspects of the Elilo loader to pass EFI data structures to the kernel.

The changes to the IA-32 kernel initialization sequence provide the capability to initialize kernel data structures, such as the memory manager, using EFI tables and data structures versus those obtained via BIOS calls. Primarily isolated in the architecturally specific directory of the kernel source tree, EFI support provides additional initialization options without affecting existing functionality. In other words, the changes to the kernel initialization sequence do not radically alter the architecture of the kernel.

Inclusion of UGA support necessitates several new Linux kernel drivers; however, this merely provides an additional console display option. Collectively, these drivers serve as an alternative to the legacy VGA console driver, but may also operationally coexist.

4.3 Architectural Influence

Itanium Linux kernels have included support for EFI for several years. Accordingly, much of the design discussed in this paper for IA-32 kernels has been leveraged from the Itanium port of Linux. Additionally, the prototype implementation has reused EFI related code to also support IA-32.

5 EFI Linux Boot Loader

Launching an operating system from EFI requires an EFI aware boot loader. This section provides background on the Elilo Linux Boot Loader, design considerations for improved IA-32 support, and details of the new boot parameters structure used to convey platform configuration information to the kernel.

5.1 Elilo Background

Elilo is the predominant loader used to launch Linux on Itanium platforms (on which EFI is the only pre-boot firmware solution) and has been included in numerous Itanium specific Linux distributions. Despite the pointed focus on supporting the Itanium architecture, a framework for booting self-extracting compressed IA-32 kernels has been incorporated into Elilo. This support permits booting IA-32 kernels without passing EFI information to the kernel. Instead, there is an implicit assumption that legacy mechanisms, such as BIOS calls, still exist for traditional platform configuration information retrieval. Essentially, Elilo manually fabricates the legacy boot parameters data structure without any semblance of EFI awareness.

5.2 Elilo Design Considerations

The primary consideration for modifying the Elilo loader is to provide adequate EFI information to the kernel to ensure proper initialization and functionality in a legacy free environment. The information required by the kernel consists of:

- Kernel Location and Size
- RAM disk (initrd) Location and Size
- Kernel Command Line
- EFI Memory Map
- Console Information
- EFI System Table
- ACPI Tables
- Other EFI Configuration Table Entries (HCDP, SMBIOS, etc.)

In addition to collecting and passing salient platform configuration information to the kernel, Elilo is also responsible for setting up the environment for passing control to the kernel. Further information regarding the features and capabilities of Elilo can be found in [6].

5.3 EFI Aware Boot Parameters

The following boot parameter structure is introduced that encapsulates the information the kernel requires. The form of the structures is as follows:

```
struct ia32_boot_params {
  UINTN command_line;
  UINTN efi_systab;
  UINTN efi_memmap;
  UINTN efi_memmap_size;
  UINTN efi_memdesc_size;
  UINTN efi_memdesc_version;
  UINTN initrd_start;
  UINTN initrd_size;
  UINTN loader_addr;
  UINTN loader_size;
  UINTN kernel_start;
  UINTN kernel_size;
  struct {
    UINT16 num_cols;
    UINT16 num_rows;
    UINT16 orig_x;
    UINT16 orig_y;
  } console_info;
} boot_parameters;
```

This data structure provides all necessary information to enable kernel initialization. Note that inclusion of the EFI system table permits access to the EFI Runtime Services and Configuration Tables that contain further platform configuration information and provide access to runtime firmware functionality. For example, the location of the ACPI tables is presented to the kernel as an entry in the EFI Configuration Table.

6 EFI Kernel Initialization

The Linux kernel initialization sequence requires modification to utilize the EFI data structures that describe the platform hardware configuration. This section presents the details of the kernel modifications and contrasts these with the existing kernel initialization. The methodology for dynamic, runtime determination of the appropriate structures to use is presented as are details on the EFI support routines necessary for proper kernel initialization.

6.1 Existing EFI Kernel Initialization

The Linux kernel initialization sequence was developed to boot in 16bit real mode and obtain platform configuration information via BIOS calls. Figure 3 outlines the current initialization sequence of the Linux kernel.

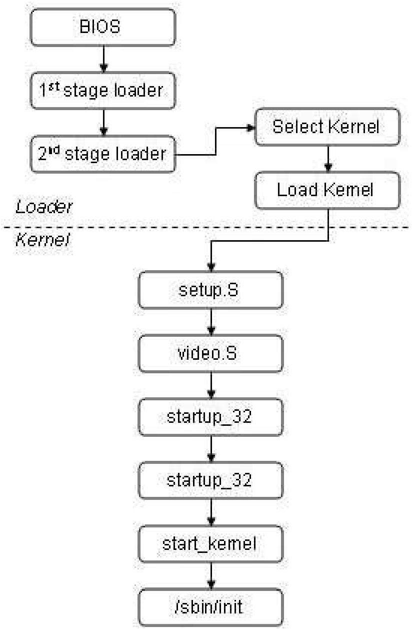

Figure 3: Existing Kernel Initialization

6.2 Kernel Initialization on EFI Platforms

Booting from EFI simplifies the kernel initialization sequence, but requires modifications to parse EFI data structures. Figure 4 depicts the proposed modified kernel initialization sequence.

In this model, the processor is already in 32 bit, protected mode when the boot loader is invoked, hence the real mode to protected mode transition code in the kernel is not necessary. Also, all platform configuration information traditionally obtained through BIOS calls is collected by the EFI Elilo loader and passed via the boot parameter structure. This alleviates the need for the code resident in setup.S,

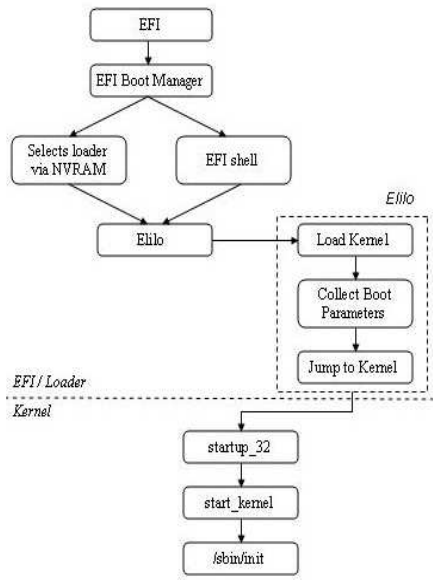

Figure 4: EFI Kernel Initialization

video.S, and bootsect.S. Consequently, control is transferred directly from the loader to the architecturally specific startup_32() routine in 32 bit, protected mode.

6.3 Dynamic Configuration Detection

Because the structure of platform configuration information on EFI platforms differs from the structure of BIOS provided information, the kernel must determine which to use to initialize kernel data structures. This determination is based on the location of the boot parameters in memory.

The boot parameters of legacy kernels are placed in a designated page in memory. Leveraging the re-use of this memory area, the boot parameters of EFI aware kernels are placed within the same page, but at a different offset. The correct kernel initialization code path to follow is determined by verifying which boot parameters have been passed to the kernel. A new global flag `efi_enabled` is set if the kernel was found to be loaded from EFI. This flag is used during the kernel initialization sequence to determine the appropriate data structures to use during initialization. This capability enables a single kernel image to load on platform with either legacy BIOS or EFI.

6.4 EFI Data Structure Mapping

Initially, only a limited kernel virtual address space is available through the temporary, statically initialized page global directory `swapper_pg_dir`. This maps the first eight megabytes of physical memory to kernel virtual address space starting at 3GB.

This limited mapping is not sufficient because EFI drivers and related structures may be located anywhere in memory (below 4GB), thus the kernel requires the capability to dynamically map EFI pages into kernel virtual address space for use during kernel initialization. Examples of these structures include the EFI memory map, RAM disk details, etc. Because these structures are only used during early initialization, the memory is only required temporarily. Once the kernel memory manager is initialized these memory regions will be available as for normal kernel memory allocations.

6.5 EFI Memory Map

The EFI memory map provides a snapshot of system memory usage before control is passed to the kernel. Consisting of memory descriptor entries that describe contiguous ranges of physical memory by type, attribute, and size, the EFI memory map serves as a replacement for the e820h memory map obtained via the legacy INT 15h (ax=0xe820) BIOS call. Each EFI memory map descriptor consists of the following structure:

```
struct efi_memory_desc {
  u32 type;
  u64 phys_start;
  u64 virt_start;
  u64 num_pages;
  u64 attribute;
};
```

In order to properly initialize the kernel memory manager as well as the EFI Runtime Drivers and Services, the EFI memory map is required. Several routines from the Itanium kernel have been massaged into the IA-32 kernel to support EFI memory map traversal and parsing.

6.5.1 EFI Memory Map Walking Routine

The prototype for the EFI memory map descriptor traversal routine is:

```
void efi_memmap_walk(
        efi_freemem_callback_t
        callback, void *arg);
```

This routine employs a callback mechanism used to discern further information about memory regions during memory map traversal. For example, used in concert with the find_max_pfn() callback routine, the kernel is capable of discovering the maximum page frame number.

6.5.2 Other Memory Map Related Routines

Several additional routines have been added to accommodate initialization using the EFI memory map. The following routine is used to determine the memory type of the region described by a given memory descriptor.

```
static int is_available_memory(struct
                 efi_memory_desc *md);
```

Differentiation between memory descriptor types is necessary to determine usable regions by the kernel. The following types constitute memory regions available for kernel use.

- EFI_LOADER_CODE
- EFI_LOADER_DATA
- EFI_BOOT_SERVICES_CODE
- EFI_BOOT_SERVICES_DATA
- EFI_CONVENTIONAL_MEMORY

The following two functions are convenience functions used to determine the type and attributes of memory regions in the EFI memory map given an address.

```
u32 efi_mem_type(unsigned long
                 phys_addr);
u64 efi_mem_attributes(unsigned long
                       phys_addr);
```

6.6 Persistent EFI Drivers and Services

Unlike EFI Boot Services, which are terminated when control is transitioned to the kernel, several EFI drivers and services are available for use during OS runtime. The following sections detail these services and describe the support framework for maintaining access to these drivers and services as well as managing the mappings into kernel virtual address space.

6.6.1 EFI Runtime Drivers and Services

The IA-32 Linux kernel requires the capability to call the EFI Runtime Drivers and Services to take advantage of platform specific functionality, such as access to the real time clock (RTC), persistent EFI NVRAM environment variables, and the capability to reset the system. The following structure, included in the include/linux/efi.h header constitutes the kernel data structure through which calls to the EFI Runtime Services and Drivers are managed.

```
struct efi_runtime_services {
  struct efi_table_hdr hdr;
  unsigned long get_time;
  unsigned long set_time;
  unsigned long get_wakeup_time;
  unsigned long set_wakeup_time;
  unsigned long
    set_virtual_address_map;
  unsigned long convert_pointer;
  unsigned long get_variable;
  unsigned long get_next_variable;
  unsigned long set_variable;
  unsigned long
    get_next_high_mono_count;
  unsigned long reset_system;
};
```

Additional EFI Runtime Drivers may be employed to exploit OS independent functionality. For example, the floating point software assist driver (fpswa.efi) on Itanium platforms discussed in [4] and the Universal Graphics Adapter are EFI compliant runtime drivers used on both IA-32 and Itanium platforms. Details of runtime drivers are passed to the operating system via the EFI Configuration Table.

The memory occupied by runtime drivers is reserved by the kernel to prevent the memory manager from viewing the area as free memory. Additionally, these memory regions are mapped into kernel virtual address space to avoid the overhead of invoking these services in flat, physical addressing mode. The mapping of runtime services and drivers into kernel virtual address space is provided by the following routine:

```
void efi_enter_virtual_mode(void);
```

This routine walks the EFI memory map and maps all regions described by memory descriptors of the following type:

- RunTimeServicesCode
- RunTimeServicesData

Once mapped, the VirtualStart field of the memory descriptor is updated with the virtual addressed returned by the mapping function, `ioremap()`. After all memory descriptors have been updated with virtual addresses, the EFI Runtime routine SetVirtualAddressMap is invoked and passed the updated EFI memory map. SetVirtualAddressMap must be called in physical mode requiring the capability to transition to physical mode before invocation and return. This function updates all EFI runtime images with virtual addresses and completes all necessary fix-ups to enable EFI Runtime Services to be called in virtual mode. Should the call to SetVirtualAddressMap fail to complete or return an error status code, the kernel will panic.

During the mapping process, each memory descriptor is also checked to determine if the address for the EFI system table is included in the range. Once SetVirtualAddressMap returns, the EFI System Table pointer is updated with the newly assigned kernel virtual address as are as the kernel's EFI data structures.

Because the ioremap capability is not available until the kernel memory manager is initialized the efi_enter_virtual_mode() function must be called after the mem_init() and kmem_cache_sizes_init() functions in start_kernel().

6.7 ACPI Initialization

In order to support device discovery and power management, kernel support for ACPI is required. The ACPI tables contain vital platform configuration information necessary for proper kernel initialization, such as the number of processors in a system, PCI interrupt routing, etc. Inclusion of ACPI support in kernel builds requires the kernel configuration flag CONFIG_ACPI_EFI to be defined in order to enable inspection of the kernel's EFI data structure for the ACPI tables. An additional requirement for proper ACPI table discovery is to update the following ACPI initialization function:

```
unsigned long __init
         acpi_find_rsdp(void);
```

On legacy systems this function scans memory looking for the Root System Description Pointer (RSDP). On EFI based systems, the address of the ACPI tables is included in the EFI Configuration Table. This function has been updated to examine the EFI Configuration Table for address of the RSDP.

7 Kernel UGA Architecture

The Linux kernel requires UGA support in order to provide console display functionality on legacy free platforms. Because the UGA is compiled into EFI Byte Code and is programmatically designed for execution in the EFI environment, kernel level support for UGA requires the addition of new driver functionality. These drivers, also pictorially described in figure 5, include:

- EFI Boot Service Emulation Driver
- EBC VM & Interpreter Driver
- UGA Console Driver

Each of these components may be built as kernel driver modules, although all are required for proper console display. Because these drivers have minimal architectural dependencies, all EBC drivers are included in a new `drivers/ebc` directory of the kernel tree.

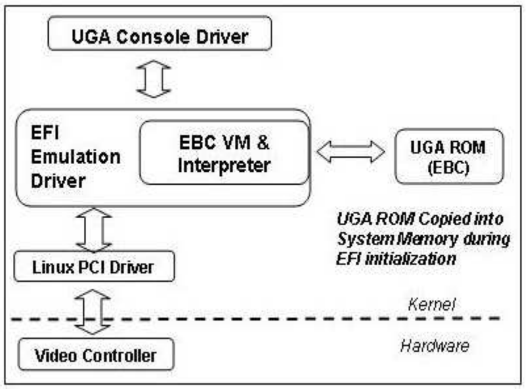

Figure 5: Kernel UGA Architecture

7.1 EFI Boot Services Emulation Driver

As is the case with all EFI drivers, the UGA requires the EFI system table, as well as a pointer to itself, as initialization parameters. All requests from the UGA for services to allocate memory, bind to the respective controller, and other routines are based on EFI Boot Services functionality. Because EFI Boot Services are terminated when ExitBootServices() is invoked, a minimal framework must be fabricated within the kernel in order to simulate EFI Boot Services. The Boot Services Emulation Driver fulfills this requirement by providing kernel implementations of the Boot Service routines. For example, the EFI memory allocation routine AllocatePages() is implemented with the kernel's kmalloc() service.

7.2 EBC VM & Interpreter Driver

A kernel implementation of an EBC Virtual Machine and interpreter is also required. This driver provides the framework and execution engine to:

- Interpret and execute all EBC instructions
- Provide an interface to handle calls between a native environment and the VM
- Provide an interface to fix-up calls to EBC driver images.

Details on the full EBC instruction set can be found in Chapter 19 of [1]. However, details on two instructions that require particular attention are included in the following sections.

7.2.1 BREAK 5 Instruction

The BREAK 5 instruction enables the transition of the instruction stream from native to EBC instructions. This technique is referred to as thunking in [1]. During compilation of the UGA ROM image, the EBC compiler inserts BREAK 5 instructions in the initialization instruction sequence to handle the transition for each of the image's protocol entry points. Functionally, the BREAK 5 instruction introduces a level of indirection, as the VM/Interpreter must replace the address of every entry point in an image with the address of a thunk. The thunk is an area in memory that includes the hexadecimal encoding of native instructions to transition control to the interpreter with an entry point of the image. This enables the seamless interpretation and execution of the EBC instruction stream via the UGA protocol interfaces. Figure 6 depicts the logical conceptual calling mechanism and an example implementation used to invoke the VM and interpreter at the appropriate UGA protocol entry point.

7.2.2 CALLEX Instruction

The CALLEX instruction provides the mechanism to facilitate calls outside the context of the EBC instruction stream or VM. For example during compilation, when the compiler observes a call to a Boot Service function, it inserts a CALLEX instruction, such that the tran-

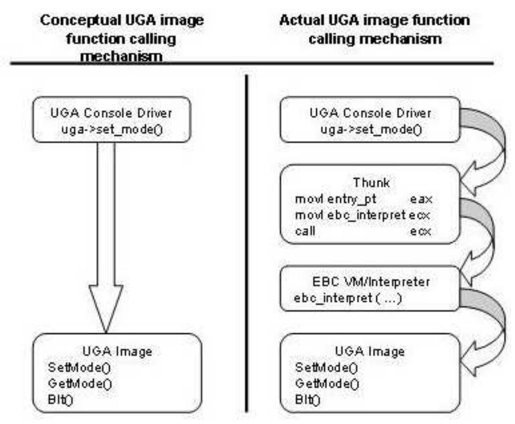

Figure 6: Thunking Mechanism

sition of the stack, IP, return value, etc. are handled gracefully.

7.2.3 Architectural Dependencies

This driver supports both the IA-64 and IA-32 architectures. Architecturally specific, required features are included through inclusion of appropriate header files. For example, the inline assembly routines used to manipulate the stack pointer and obtain the entry point from a processor register are included in architecturally specific headers.

7.3 UGA Console Driver

The UGA console driver provides an interface between the Linux kernel console subsystem and the UGA ROM. Because the UGA is conceptually encapsulated (i.e. registered) with the EFI Boot Services Emulation driver, the console driver will affect changes to the display through the use of the UGA_IO_PROTOCOL and UGA_DRAW_PROTOCOL protocols. As a result, the console driver must obtain the UGA protocol interface structures from the EFI Boot Services Emulation Driver.

Unlike VGA, multiple UGAs may be used in a single system. Therefore, the UGA console driver must maintain data structures to handle multiple UGA simultaneously. For early boot message display, the console driver will initially employ the UGA used by the firmware, the details of which are included in the boot parameters.

Similar to the VGA console driver, the UGA console driver supports the generic console routines through the `consw` structure. Once the console driver obtains the protocol interfaces from the EFI Emulation Driver and the driver is initialized the UGA may be used for console display.

7.3.1 Firmware to OS Handoff Structure Parsing

Details of the UGAs discovered during firmware initialization are passed via the EFI Configuration Table. Each entry in the Configuration Table consists of a GUID and pointer pair. During kernel initialization, the pointer in the Configuration Table is stored in the kernel's efi structure. In the case of UGA, this points to the firmware-to-OS handoff header, which is of the following form:

```
struct efi_os_handoff_hdr {
  u32 version;
  u32 hdr_size;
  u32 entry_size;
  u32 num_entries;
};
```

This header is immediately followed by the driver handoff entries for all UGAs discovered by the firmware. Each entry driver handoff structure is of the following form:

```
struct efi_driver_handoff {
  int type;
  struct efi_dev_path *dev_path;
  void *pci_rom;
  u64 pci_rom_size;
};
```

The UGA Console driver parses each of these driver handoff structures to obtain device information as well as the address of the PCI Expansion ROM that has been copied into memory by the firmware. The PCI Expansion ROM is then parsed to locate the EBC UGA image. Once the UGA image is located, the console driver updates its internal data structures with the image address and device information. Figure 7 shows the organization of the information passed from the firmware to the kernel.

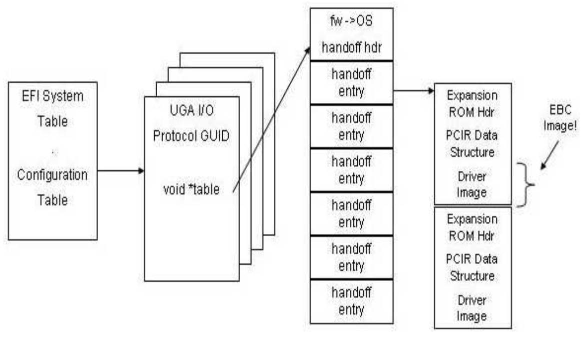

Figure 7: UGA Configuration Table Parsing

Details on the header information and layout of the ROM image may be found in [2].

7.3.2 Executable Image Parsing

After the EBC image is located, the console driver initializes a list of UGAs in the system with device information as well as pointers to the actual PE32+ UGA EBC image in memory. In order to use the driver, the image must be effectively loaded. Although the image is already in memory, the image must be parsed to correctly identify the entry point and image specific information. Once the entry point is obtained, the EBC VM and Interpreter is invoked through the EFI Boot Services Emulation Driver and the UGA is initialized. The effective loading and parsing of PE32+ EFI images requires PE header structure information be included in the kernel.

8 Future Challenges

The possibility exists to offload kernel decompression to the Elilo loader. This loader extension would provide the capability to boot both compressed and uncompressed IA-32 kernel images, similar to existing functionality on Itanium platforms. This is an area that is still under investigation.

Additional work to enable the use of the more advanced features of UGA is ongoing. A prototype implementation has been developed that supports the UGA functionality discussed. However, this support is currently limited to a single console and does not account for possible UGA related changes to XFree86. Further, the current implementation does not address issues involved in supporting multi-head console within the kernel through the use of UGA.

9 Conclusions

EFI provides a standard interface to platform firmware from which to launch operating systems on legacy free platforms. The IA-32 Linux kernel requires several key changes to initialize properly on EFI supported platforms. The first change involves the inclusion of additional IA-32 support in the Elilo Linux boot loader. Modification of the kernel initialization sequence to enable the use of EFI constructs such as the EFI memory map and EFI Runtime Drivers and Services are required. The kernel also requires several additional driver modules in order to use the legacy-free UGA for console display.

As platform hardware evolves and legacy hardware and legacy BIOS support is phased out, operating systems must adapt. Inclusion of the capabilities discussed in this paper constitutes a significant step towards enabling Linux to boot on EFI based, legacy free platforms.

10 Acknowledgements

Special thanks to Mark Doran, Andrew Fish, Harry Hsiung, Mike Kinney, and Greg McGrath of the Intel EFI team for their contributions to this paper and always helpful advice. Thanks to Steve Carbonari, Mark Gross, Tony Luck, Asit Mallick, Mark Gross, Mohan Kumar, Sunil Saxena, and Rohit Seth for reviewing this paper.

Special credit is due to Mark Gross for his efforts spent working on the design and implementation of the prototype EFI enabled kernel.

References

[1] *EFI Specification Version 1.1, Intel Corporation, 2003* `http://developer.intel.com/technology/efi`

[2] *PCI Local Bus Specification,* Revision 2.3, PCI Special Interest Group, Hillsboro, OR `http://www.pcisig.org/specifications`

[3] Werner Almesberger, *Booting Linux: The History and the Future* Proceedings of the Ottawa Linux Symposium 2000, July 2000

[4] David Mosberger and Stephane Eranian, *ia-64 Linux Kernel, Design and Implementation.* Prentice Hall, NJ 2002.

[5] *ACPI Specification* Version 2.0b `http://www.acpi.info/spec.htm`

[6] David Mosberger and Stephane Eranian, *elilo-3.3a Documentation.* 2000. `ftp://ftp.hpl.hp.com/pub/linux-ia64/elilo-3.3a.tar.gz`

[7] Werner Almesberger, *Lilo Technical Overview,* `ftp://metalab.unc.edu/pub/Linux/system/boot/lilo`

[8] Eric Boleyn, et al. *GNU Grub* `http://www.gnu.org/software/grub/grub.html`

[9] *Wired for Management Baseline,* Intel Corporation, 1998. `http://developer.intel.com/ial/WfM/wfmspecs.htm`

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada