WrtVMM: A Virtual Machine Monitor for Embedded Devices

Aaron Gember
agember@cs.wisc.edu

Yueh-Hsuan Chiang
yhchiang@cs.wisc.edu

Xixuan Feng
xfeng@cs.wisc.edu

ABSTRACT

We built a virtual machine monitor for the Linksys WRT54GL wireless router to run Embedded Xinu as a guest OS within OpenWrt. The system uses a kernel module and signal handlers to support the virtualization of the guest OS. Sufficient support is provided to allow Xinu to startup, handle timer interrupts, provide I/O, and execute processes in the Xinu shell. The virtual machine performance for processor and memory intensive tasks is similar to the execution time when running the two operating systems alone. Network throughput is unaffected by the VMM.

1. INTRODUCTION

Embedded devices are becoming smaller, faster, and more powerful with every passing day. This increase in embedded hardware performance is most evident in the mobile phone market, but also extends to other embedded platforms. Embedded devices ranging from global positioning systems to printers and Blu-Ray players to home network routers are providing more functionality for their users. Developers are challenged to add new services to a diversity of devices and still keep the core services functioning as expected.

Some techniques have already been implemented to provide a framework for developing hardware independent services. Use of an embedded Java Virtual Machine (JVM) is a common solution used by embedded application developers to write code that can be run on a diverse set of hardware. The shortcoming of the JVM is the overhead of using just-in-time compilation on an already resource constrained system. Other vendors provide a high-level application API for their devices, but the great variability in APIs requires developers to customize their solutions for each platform. There remains a need for providing a device independent framework for embedded hardware and protecting the low-level systems on the devices.

Virtualization is a long standing technique that has been successfully deployed in the PC and server market to address the needs of device independence and preservation of core services. Virtual machines (VMs) allow developers to setup applications and services in a hardware independent manner, making it easy to clone and migrate the services to various devices. In addition, isolation of services within a virtual machine prevents errant applications from interfering with applications running outside the VM. These benefits of flexibility, ease of use, and security make things easier for administrators and developers and hopefully provide a better experience for the user.

Since virtualization has seen such success in the PC market, we decided to apply the same techniques to embedded devices. Our goal was to build an embedded virtual machine monitor (VMM) for the Linksys WRT54GL wireless router [12] and run Embedded Xinu [7] as a guest operating system. The approach we use is known as paravirtualization because some modifications are made to the guest OS to allow it to run as a virtual machine. (Full virtualization proved too complex for the time available.)

We selected home networking hardware as our embedded platform of choice because of its low cost and wide availability. Home networking hardware is an embedded device which has the potential to serve more purposes than just processing network packets. These devices can be used as print servers or network file servers, while still routing network traffic. We chose to virtualize an entire operating system (rather than just a part of an application interface) because it provides more flexibility and greater isolation between services. Embedded Xinu can provide applications with all the standard operating system primitives including process management and communication, memory management, devices, networking, and a user shell. Our VMM supports system bootstrapping and startup, privileged instruction emulation, interrupt handling, simple memory management, and simple I/O. We have not included network virtualization in the scope of this project because of its complexity.

Using virtual machines in wireless routers can have some of the same benefits of their desktop-based counterparts. Building an embedded VMM for embedded networking hardware has three primary motivations:

1. **Isolation between core networking functionality and additional services:** As more services— print server, network file server, etc.— are added to the router, the risk of instability increases. Errors in one of the additional services could lead to the degradation or failure of the networking services the router provides. Additional services should be run in a virtual machine to protect networking routing from being negatively affected.
2. **Ability to move and clone services between routers:** Some network services can take a significant amount of effort to configure. Therefore, the ability to easily move a service and its configuration between systems

can be a major benefit. A virtual machine allows a user to easily move services between routers, clone the service across multiple routers, or share the setup with other interested users. Good examples of services that could benefit are a Squid proxy server or OpenLDAP server.

3. **Elimination of additional hardware for simple network-based services:** Rather than requiring a desktop computer to be constantly running, simple network-based services can be run on the router. Print servers and network file servers (for routers with USB ports) are already popular. But other services like a Voice over IP phone, a small HTTP or LDAP server, a Squid Proxy server, or simple email relay agent could be run on a router. Running the service inside a virtual machine allows a configuration similar to the desktop equivalent.

Throughout the paper we use the terminology "Xinu alone" to refer to the execution of Xinu directly on the hardware without virtualization. We use "Xinu as guest" to refer to the execution of Xinu in the virtual machine running on top of OpenWrt.

The rest of this paper is organized as follows. Section 2 discusses the hardware and software platforms used. An overview of our system architecture is given in section 3. The implementation details of key aspects of the VMM are discussed in sections 4 through 7. We discuss challenges in section 8 and evaluate the VMM in section 9. Related work is discussed in section 10. Lastly, we conclude and discuss future work in section 11.

2. PLATFORMS

To build a VMM for an embedded device, we needed to select three core components: an embedded hardware device, a host OS, and a guest OS. Since the project was partially inspired by one of the author's prior experience with Embedded Xinu, this simple operating system was selected as the guest OS. As consequence, the Linksys WRT54GL wireless router became a natural choice for the hardware platform. Lastly, OpenWrt was selected as the host OS because of its Linux base and easy to use development environment.

2.1 Linksys WRT54GL (Hardware)

The Linksys WRT54GL wireless router is a widely available home networking device sold in most electronics stores for about $55. The router contains one of Broadcom's BCM47xx family "system-on-a-chip" with a 32-bit, 200MHz embedded MIPS processor [6]. It is equipped with 16MB of RAM and a memory management unit which relies on a translation lookaside buffer (TLB) for memory address mappings. Four MB of flash memory provides persistent storage on the device. For networking, the router features one WAN port, four LAN ports, and 802.11 wireless. The hardware's low price and broad Linux support make it an ideal platform for developing an embedded VMM.

Some minor modifications to the hardware – the addition of two serial ports – allows us to have fuller access to the router's internals. (Details on adding the two serial ports are available from [5].) Using a desktop computer and a serial connection, we are able to access the Common Firmware Environment (CFE) on the router. CFE provides a mechanism for downloading and applying a boot image to the router and bootstrapping the host OS startup. The WRT54GL is a well equipped embedded device, but it still exemplifies the challenges associated with embedded systems.

2.2 OpenWrt (Host OS)

Ever since users first discovered Linksys was running Linux on the routers in the WRT54G family, numerous open source Linux distributions have been target towards this hardware. We considered DD-Wrt [1], Tomato, and FreeWrt, but selected OpenWrt for its well designed and easy to use development environment. The latest version of OpenWrt uses the 2.6.30 version of the Linux kernel. The build environment automatically downloads the kernel source and applies a set of patches to allow the kernel to run on the WRT54GL. The build system also builds the necessary cross-compiler tools – a difficult task typically deemed to require some form of magic.

OpenWrt features a package system which allows for the inclusion of additional libraries and software in the operating system. The uClibc C library included with OpenWrt makes it easy for developers to run existing Linux applications (with some minor patching) or develop new software. The packages, a base filesystem, and the Linux kernel are compiled into a single binary file which is stored in the router's flash memory using TFTP and CFE. Once OpenWrt is installed on the router, users can use the serial ports to communicate with the BusyBox shell and work with the system.

2.3 Embedded Xinu (Guest OS)

The Embedded Xinu operating system was first developed at Purdue University in the mid-1980s. Most recently, Xinu has been targeted towards embedded networking hardware from Linksys. Xinu, which stands for "Xinu Is Not Unix," is not based on Unix or Linux. The OS is designed for research and educational purposes, with simplicity in mind. Xinu provides most of the standard functionality of today's operating systems: process management, memory management, I/O, networking, and a user shell. The OS lacks a filesystem and memory protection is in its infancy, but this makes it no less desirable of a system. In fact, this simplicity is what makes Embedded Xinu a perfect choice for our guest OS.

3. ARCHITECTURE

Figure 1 shows the four main components that make up our system: the OpenWRT Linux kernel sits on the hardware, the VMM module is installed into the kernel, the vmm-launch process and Embedded Xinu are in user space.

We are using the standard OpenWRT Linux kernel, minimizing and even eliminating modifications of its source. By doing this, we do not require user to have knowledge of the Linux kernel source to use the system. The Linux kernel module is a very important mechanism in the system. The VMM module provides necessary communication between the host OS and guest OS.

The vmm-launch component prepares the environment for the guest OS to run, including some signal handlers that receive notifications from the host OS or the VMM module. Slight modifications have been made to Xinu to allow it to operate correctly in the virtualized environment. From OpenWRT's point of view, Xinu runs under the vmm-launch process in user space.

4. MEMORY MANAGEMENT

The first challenge running Xinu as a guest OS is memory. As an operating system, Xinu expects to stay at kernel segment, but OpenWrt already resides in the kernel segment. In addition, we need some executable memory in which to place the Xinu code.

4.1 Embedded Xinu Memory Management

Memory on MIPS-based processors is divided into four segments, which includes one user segment and three kernel segments.

4.1.1 User Segment

The user segment of memory, *USEG*, ranges from `0x00000000` through `0x7FFFFFFF`. This memory is both mapped and cached. Any attempted access in this segment must with the CPU privilege level set to user mode or the processor must be in the exception state with the error level bit set. When a TLB exception occurs in the USEG range of addresses, a special fast exception handler, at `0x80000000`, is consulted instead. Exception code to handle TLB faults simply performs a lookup of the faulting address in the system page table, checks to see if it is valid and in the correct address space, and inserts the mapping in the TLB hardware. We have disabled the use of user segment memory and memory protection in the version of Xinu we are running as a guest.

4.1.2 Kernel Segment

While the first part of memory is dedicated to the user segment, the remainder is the kernel segment. Unlike the user segment, the kernel segment is sub-divided into three segments with different memory access properties. They are KSEG0, KSEG1, and KSEG2.

KSEG0 is the range of memory addresses from `0x80000000` through `0x9FFFFFFF`; it is unmapped and uncached. Embedded Xinu exclusively uses this segment. A generic exception handler is loaded at `0x80000180`. Xinu also makes use of reserved memory starting at `0x80000200` to store an array of exception handler entry points and `0x80000280` to store an array of interrupt handler entry points. As for loading the kernel, Xinu loads the kernel code beginning at `0x80001000`.

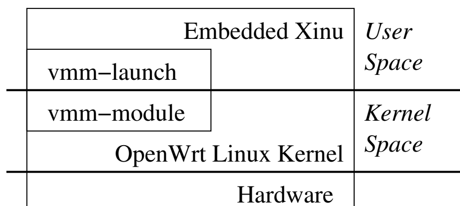

Figure 1: System architecture

Upon booting, CFE transfers execution control to that address. Xinu uses the remaining memory from this segment as the user memory heap. Once this memory is initialized, calls to malloc and free will use the user heap for memory allocation and automatically insert mappings into the system page table. All mappings are 1-1 since there is no backing store for a virtual memory subsystem.

KSEG1 is the range of memory addresses from `0xA0000000` through `0xBFFFFFFF`. This memory is unmapped and uncached. Embedded Xinu uses KSEG1 to to access some hardware devices which are mapped out-of-range of physical memory and some hardware devices use dynamically allocated memory for sharing. Such devices on WRT54GL include the serial ports (or UART), the Ethernet hardware, and the Wireless LAN.

KSEG2 is the range of memory addresses from `0xC0000000` through `0xFFFFFFFF`. This memory is both mapped and cached. Embedded Xinu does not make use of any KSEG2 memory.

4.2 VMM Module

To enable embedded Xinu to run as a guest OS, the VMM module, a Linux kernel module, is introduced into our system. Linux kernel modules are a widely-used mechanism to implement device drivers which contain several fundamental driver operation prototypes. Once these operations of the VMM module have been carefully designed and specialized, the VMM module is able to provide its operations and enable Xinu to run as a guest OS.

The following lists some important operations of the VMM module:

- **mmap()** is used to request a mapping of device (or kernel) memory to a process's address space. We utilize this operation to allocate executable memory and map it to the vmm-launch process's address space where our guest OS runs. When the `mmap` function of the VMM module is called, a starting memory address is passed as an input parameter. The VMM module allocates executable memory and performs page-wise memory mapping to map the memory pages into the user process's memory address space.

  When the guest OS is shut down, the memory unmapping request will be invoked and the `unmap` function of the VMM module will be called. The allocated executable memory will be unmapped and released.

- **ioctl()** offers a way to issue device-specific commands. We utilize this operation to request secondary interrupt handlers for notifying our guest OS of interrupts. We discuss this in more detail in section 6.

4.3 Memory allocation for VM

As we mentioned in the architecture section, the vmm-launch process serves as a medium between the guest OS and the VMM module. When the vmm-launch process is started it calls the `mmap` provided by the VMM module with the starting address of the guest OS passed as an input parameter. Then, the VMM module will allocate executable memory and perform page-wise memory mapping to map it to the

user space memory starting at the desired address specified by the input parameter. After the `mmap` is completed, the memory region beginning at that starting address is mapped to pages of executable memory, and vmm-launch will load the Xinu code into that memory space. One MB of memory (1024 4 KB pages) are allocated for Xinu as guest. The memory is mapped into user space beginning at `0x30000000`. The Xinu linker script is modified to compile a Xinu image that can be loaded and executed beginning at `0x30001000` instead of the usual `0x80001000`.

In addition to the memory region used for the code and stack of Xinu as guest, a small piece of memory is allocated in the same way to achieve CPU virtualization. We refer to this special page of memory as the *virtual CPU*, or VCPU. This memory stores virtualized hardware data structures such as control registers. This virtual CPU is also supported by the `mmap` operation of VMM module. We will explore how we use this the VCPU to handle privileged instruction in the next section.

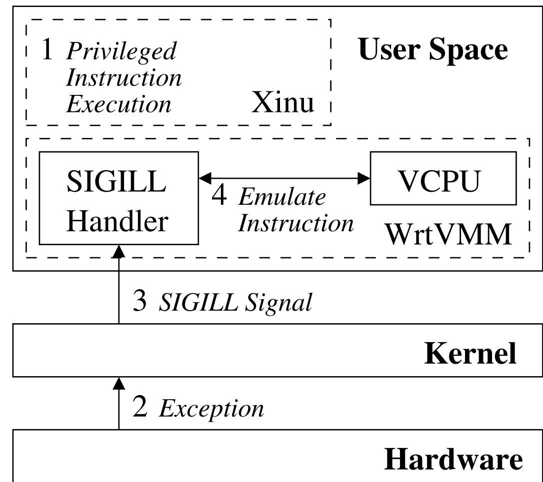

Figure 2: Privileged instruction handling

5. PRIVILEGED INSTRUCTIONS

As an operating system, Xinu is responsible for low-level control of the physical hardware. During system startup, for example, Xinu's first task is to flush the instruction and data caches. Xinu reads from a special purpose register to determine the cache sizes and uses the MIPS assembly `cache` instruction to fill the caches with zeroes. Other examples of low-level control include exception handling, interrupt control, and clock management. When running Xinu alone, the OS has full reign over physical hardware.

OpenWrt is responsible for low-level control of the physical hardware when running Xinu as guest. The Linux kernel has full reign over physical hardware and the processor is placed in *kernel-mode*. When Linux schedules a user process to execute, such as Xinu as guest, the processor is switched to *user-mode*. OpenWrt restricts user processes from accessing special purpose registers – referred to as coprocessor 0 (CP0) registers – and executing special assembly instructions – referred to as *privileged instructions*. Since the Linux kernel is already controlling the physical hardware, we want Xinu as guest to control virtual hardware. The challenge is to make Xinu control virtual hardware without modifying the code designed for controlling physical hardware.

5.1 Illegal Instruction Signal (SIGILL)

If code attempts to execute a privileged instruction while the processor is in user-mode, the hardware throws an exception. The processor stops execution of the current code and jumps to an exception handler at a known memory location. The Linux exception handler stores the current values of the general purposes registers and the program counter. A SIGILL signal is sent to the user process that was executing when the exception occurred. When the user process resumes execution, it receives the SIGILL signal and, by default, terminates the user process. When starting Xinu as guest with the default signal handling behavior, Xinu will execute a privileged instruction to flush the caches, resulting in an exception and process termination.

We register a custom SIGILL signal handler to transform Xinu's attempted control of physical hardware into control of virtual hardware. The same process still occurs when Xinu as guest executes a privileged instruction, but instead of process termination, our signal handler is triggered by the SIGILL signal. The custom signal handler receives a structure containing the execution context (values of general purposes registers and the program counter) when the privileged instruction was attempted. We use and modify the execution context and the special memory allocated to the VMM to emulate the instruction and make Xinu as guest control virtual hardware. After signal handling, the execution context is restored and normal execution of Xinu resumes. Figure 2 shows the entire privileged instruction handling process.

5.2 Instruction Emulation

The execution context provided to our SIGILL signal handler contains the address stored in the program counter prior to the illegal instruction exception. The address will be within the Xinu code, loaded into memory as discussed in section 4. We obtain the 4-byte encoding of the faulting instruction from that address. The opcode is identified using bitwise operators. We currently recognize four opcodes: move to coprocessor 0 (`mtc0`), move from coprocessor 0 (`mfc0`), modify cache (`cache`), and jump (`j`). The first three opcodes are privileged instructions. Jump is a special case when the instruction is `j ra` (jump to the address stored in the return address register), and the instruction is directly followed by a `mtc0` or `mfc0` instruction; this special case is a result of the `mtc0` or `mfc0` instructions taking two processor cycles to complete.

After determining the faulting opcode, the appropriate behavior is emulated to control our virtual hardware. Since our virtual machine has no explicit caching mechanisms, the `cache` instruction is simply ignored. For a `j ra` instruction, we check if a `mtc0` or `mfc0` instruction follows, and perform the emulation for those two instructions. A `mtc0` instruction is emulated using the following steps:

1. Determine the number of the general purpose register which contains the value to store.

2. Determine the number of the coprocessor 0 (CP0) register to which the value should be stored. If we do not keep track of the CP0 register being referenced, skip to the end of the emulation process.
3. Obtain the value in the general purpose register from the execution context.
4. Store the value in the appropriate memory location within the VCPU based on the CP0 register number.

The `mfc0` instruction is emulated using a similar set of steps.

The final task in the SIGILL signal handler is to update the program counter in the execution context. After emulating the instruction, we want to resume execution at the instruction just after the privileged instruction. For `mtc0`, `mfc0`, and `cache` instructions, the program counter is incremented by four. For the `j ra` instruction, the program counter is set to the value of the return address register provided in the execution context. The signal handler completes and execution of Xinu as guest resumes with the modified execution state.

5.3 Limitations of Signal Handlers

Using a custom SIGILL signal handler to emulate privileged instructions works well in most cases, but there are two limitations of this method. First, not all instructions we need to emulate result in a SIGILL signal. The system call instruction (`syscall`) is currently used for virtualizing I/O, but in the future it should properly be emulated to allow system calls within Xinu. Since `syscall` is not a privileged instruction, no SIGILL signal is generated and the instruction will not be trapped for emulation.

The second limitation of our method stems from a limitation of the Linux kernel: only one signal handler can be executed at a time. A problem occurs when a privileged instruction must be executed as part of an interrupt handler, which also uses a signal handler. The interrupt handling code is already in the middle of a signal handler, so any privileged instruction that is executed during this time will result in process termination instead of calling the custom SIGILL signal handler. Using multiple threads is one possible solution to the problem, but the Pthreads library in the development version of OpenWrt does not function correctly. The solution we adopt is to modify Xinu code and replace the privileged instructions. A `mtc0` instruction is replaced with two instructions: load the address of the CP0 register in the VCPU and store the value at that location. The `mfc0` instruction is the similar. This solution is undesirable because it requires modifications to the guest OS.

6. INTERRUPT HANDLING

The VMM must support interrupts for Xinu as guest to run correctly. At the very least, we need timer interrupts to enable scheduling with preemption. The challenge, again, is to make Xinu receive interrupts from virtual hardware, since OpenWrt is already responsible for processing interrupts received from the physical hardware.

6.1 MIPS Interrupts

Interrupts are enabled and disabled by setting the appropriate bits in the CP0 cause register. There is a bit for each of the 8 IRQ numbers. In the WRT54GL, IRQ 3 is the serial ports and IRQ 7 is the hardware clock timer. The register also has a master enable bit to enable or disable all interrupts. Xinu has functions to enable and disable specific IRQ numbers and functions to enable and disable all interrupts.

When an interrupt occurs in MIPS, two main actions are taken by the processor. First, bit flags are set in the exception cause register to provide information about the cause of the interrupt. There is a bit flag for each of the 8 possible IRQ numbers. Second, execution jumps to a predefined kernel-only address (`0x80000180`), where the operating system is expected to have placed an interrupt handler. There is only space for 32 assembly instructions at the special memory location, so the OS usually does some minimal processing then jumps to a more complex interrupt handler within the kernel code.

Xinu's interrupt handling mechanism is composed of multiple components:

- During system startup, a simple 15-instruction handler is placed in the special memory location reserved for the interrupt handler. This piece of code is called for both exceptions and interrupts. Xinu uses bitwise operators with the CP0 cause register to determine if one of the interrupt bit flags is set and calls the `savestate` function.
- The `savestate` function, written in assembly, allocates space on the stack of the interrupted process. The value of the CP0 cause register and the values in all general purpose registers are saved on the stack. After saving the processor state, Xinu calls the `dispatch` function.
- The `dispatch` function, written in C, uses the value saved from the CP0 cause register to determine which specific IRQ number caused the interrupt. The IRQ number is used as an index into an array of interrupt vectors – functions designed to handle a specific type of interrupt. The specific interrupt handling function is called.
- In the case of a timer interrupt, the handling function updates Xinu's clock. It also updates the CP0 compare register so that another timer interrupt is fired 1 ms later. If a sleeping process has finished sleeping or a process's time quantum has expired, a new process is scheduled to run.
- Lastly, the `restorestate` functions reloads the values of the general purpose registers from the stack. It executes the `eret` instruction to return from the interrupt handler and resume executing the process which was interrupted.

6.2 Cooperating Module and Signal Handler

Since OpenWrt normally handlers hardware interrupts, we need to provide virtual interrupts to Xinu. One dilemma here is whether we should let the host OS kernel notify the guest OS of a physical hardware interrupt or not. In the case

of a timer interrupt, Xinu as guest should always also receive a timer interrupt. In the case of a serial interrupt, Xinu as guest should only receive an interrupt if the input is for the VM. To solve the issue of providing virtual interrupts, we employ a function in our VMM module and a signal handler in the vmm-launch process.

In the VMM module, we can add a binding between an IRQ number and a handler. The vmm-launch process calls the `ioctl` function in the module to initiate the binding. The module stores the Linux process id of the guest OS so the interrupt handler knows which process to signal. The Linux `request_irq` function is called to register a shared interrupt handler for the timer interrupt, IRQ 7. A shared interrupt handler means that Linux calls multiple handlers when an interrupt occurs. We made a one line modification to the Linux kernel to allow timer interrupts to be shared. All other interrupts are shared by default.

The vmm-launch process also installs a signal handler for a specific signal number, using the `sigaction` function. The signal number must be a real-time signal number to force signals to be handled in a timely manner. We use signal number 57 for the timer interrupt. The signal handler function checks if Xinu as guest has enabled interrupts by reading the value of the CP0 status register in the VCPU. If the timer interrupt is enabled, the appropriate bit is set in the CP0 cause register in the VCPU to indicate a timer interrupt has occurred. The signal handler then jumps to the standard Xinu interrupt handler, located at `0x30000180` for Xinu as guest. Xinu proceeds to handle the interrupt as normal. When the Xinu handler completes, the signal handler clears the interrupt cause bit in the CP0 cause register in the VCPU.

The entire process of interrupt handling is shown in figure 3. When a physical timer interrupt occurs, OpenWrt calls the normal Linux interrupt handler and our secondary interrupt handler. Our secondary interrupt handler sends a real-time signal to the guest OS process. The custom signal handler receives the signal, sets the appropriate registers in the VPCU and calls the Xinu interrupt handler.

7. I/O

When Xinu runs alone it relies on two devices for providing normal input and output (I/O): a UART (or serial) device and a TTY device. The UART device is responsible for directly controlling and communicating with the physical serial port hardware. The TTY device is responsible for input line buffering, formatting, and echoing; the device is typically used as an intermediary between a user application (like the shell) and the UART device.

The UART device driver is asynchronous and conceptually divided into a lower and an upper half (figure 4). Control and status registers for the two NS16550 serial ports are mapped to memory locations `0xB8000300` and `0xB8000400`. The UART device driver reads and writes control bits from eight one-byte "registers" at each of these memory locations. The registers contain the status of the hardware FIFO queue, the status of interrupts, the cause of interrupts, and other control information. One of the eight registers is read to receive a one-byte character or written to transmit a one-

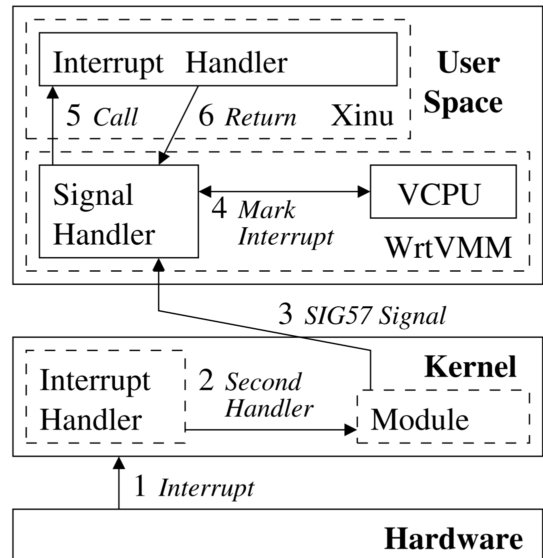

Figure 3: Interrupt handling

byte character. When an interrupt occurs, the lower-half handles the interrupt by moving received data from hardware to the inbound buffer or moving data from the outbound buffer to hardware. Data is read from or written to the two buffers when a user application makes a read or write call. The upper-half of the driver copies the data between the buffers in the driver and a user supplied buffer. For certain system I/O, Xinu uses polling to communicate with the physical hardware, but most I/O is performed asynchronously using interrupts.

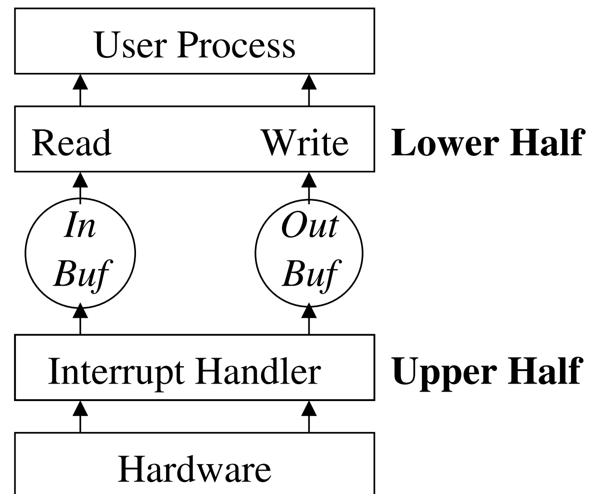

Figure 4: UART device in Xinu alone

The challenge of I/O when running Xinu as guest is sharing with OpenWrt. OpenWrt expects to use the physical serial port for I/O and have exclusive control over its functionality. The guest OS is running as a user process, so it does not have direct access to hardware or devices within the kernel. In

addition, input may be for Xinu or another process, so a decision must be made to determine to whom input should be provided.

7.1 Hypercalls

The solution is to allow OpenWrt to maintain exclusive control of the serial ports and require Xinu as guest to use standard interfaces to request I/O from the Linux kernel. Normally user processes in Linux rely on C standard library functions or the `read` and `write` system functions for I/O. Both of these methods "translate" the request into a system call. The system call number is loaded into the first argument register (a0) and the other arguments for the system call are loaded into the subsequent argument registers. On MIPS, the `syscall` instruction is executed which causes execution to jump to the exception handler at a known address (`0x80001000` for our hardware). The exception handler saves the state of the running process, looks up the system call number in a table, and dispatches to the appropriate handling function. The results of the system call are stored in the return value register (v0), the process state is restored, and the `eret` instruction is executed which causes execution to jump back to the next instruction after the `syscall`.

We modified Xinu to use the same mechanism to perform I/O. Instead of using the C standard library of Linux system functions, we created a wrapper function within Xinu that loads the appropriate system call number and arguments and executes the `syscall` instruction. The `syscall` instruction causes execution to jump to the exception handler which OpenWrt has placed at the standard memory location. We refer to this approach as a *hypercall*, because the guest OS is making a "function call" into the host OS. After the hypercall completes, Xinu will have performed I/O by relying on OpenWrt. We created a special UART device driver within Xinu, known as a uart-virtual device, which makes a hypercall in the upper half `read` and `write` functions. The device has no buffers or lower half since Xinu relies on OpenWrt to provide this functionality. The TTY device in Xinu is unmodified.

7.2 Shortcomings of Hypercalls

The hypercall mechanism provides functioning I/O within Xinu as guest, but it has two shortcomings. First, hypercalls require modifications to Xinu. We added a new UART device with a different design than the original and additional code for making hypercalls. Second, hypercalls break scheduling semantics within Xinu as guest. When Xinu alone makes a read call and no input is currently available, the process which made the request is placed on the wait queue and another process is scheduled to run. When a process running in Xinu as guest makes a read call, OpenWrt places the entire guest OS on its wait queue and another process in the host OS is allowed to run. Xinu as guest does not have a chance to schedule another process to run while a process is waiting for input.

The ideal solution is leave the UART device driver in Xinu untouched and utilize the interrupt and privileged instruction mechanisms discussed earlier to virtualize the physical serial hardware. When Xinu tries to read from the control and status registers of the serial ports, the VMM would catch the illegal memory access, emulate the operations on the registers and return control to Xinu. Xinu's interrupt handler can be called for a serial interrupt the same as we have implemented with timer interrupts. Implementing I/O without hypercalls is an important part of future work to improve performance and correct scheduling semantics when running Xinu as guest.

8. CHALLENGES

In this section we discuss some of the challenges we faced throughout the project. It's essentially the story of how we learned from failures or bad performance, and improved the system later on.

8.1 From Linking Libraries to Hypercalls

The first challenge was to be able to see some output from the virtual machine. We could not directly communicate with the serial hardware, so we attempted to use system functions within OpenWrt. Many attempts were made to link a Xinu kernel with the uClib library in OpenWrt. We tried building our own shared library, using dynamic and static libraries, and using a variety of compiler and linker flags. All methods proved unsuccessful. The eventual solution we used is that discussed in the previous section: hypercalls.

8.2 From `malloc` to the VMM Module

When we tried to allocate memory space for the guest OS, we first used `malloc` to get the memory, put some code in it and execute it. But by tracing the program, we learned that the memory from malloc can be used for executing code. We switched to building a kernel module. In the Linux kernel, `kmalloc` is used for allocation of a piece of memory that is physically contiguous, while `vmalloc` does not guarantee the physical layout. However, because of hardware constraints, `kmalloc` can allocate only up to 512KB memory, which is not sufficient for our guest OS. We decided to use `vmalloc`, but it failed again because we forgot to map the noncontiguous memory page by page. Once we fixed this problem, the guest OS initialization successfully until it tried to initialize timer interrupt, which was our next challenge.

8.3 From Calling User Space to Signals

In order to let the guest OS have interrupts, the biggest issue was how to notify a process in user space. Naturally, we decided to have the kernel module help. But even after enabling the kernel to have an additional interrupt handler for the timer interrupt, we were still in the kernel space, which means calling the Xinu handler is not possible. Signal handlers are the solution of this trouble. Before launching the guest OS, the vmm-launch process registers a signal handler for each interrupt, which can be triggered from the kernel space by sending a specific signal.

8.4 From Instruction Replacement to Emulation

Not all instructions can be executed in the user mode. At the very beginning, we sacrificed the level of virtualization by replacing such privileged instructions with direct reads from the VCPU. But after we used the signal handlers to solve the interrupt handling, we realized the kernel used signals for privileged instructions: when it captures a process trying

to execute privileged instructions, is sends a SIGILL signal to the process. Therefore, using signal handlers to emulate the privileged instructions was even easier than virtualizing interrupts.

9. EVALUATION

The evaluation of our virtual machine monitor focuses on multiple facets of performance. Most evident is the performance of our guest operating system. Xinu as guest needs to have reasonable processing and memory throughput with minimal time and space overhead. We use Xinu alone as a comparison point for evaluating the performance of Xinu as guest. We are also concerned with the performance of our host operating system. One of our goals is to be able to continue to use the core network routing functionality of OpenWrt, meaning the time and space overhead of our virtual machine needs to have a minimal affect on OpenWrt's performance. Lastly, our evaluation focuses on the VMM's ease of use and the amount of code modifications required.

9.1 Timing

It is important we first discuss the timing mechanisms used for evaluation because of their relevance to many of our performance benchmarks. We are running two different operating systems, sometimes executing as single, stand-alone systems and sometimes executing in a host/guest configuration. This diversity of configurations makes timing consistency a difficult challenge. The most accurate mechanism for timing we can use is the hardware clock, but not all evaluation configurations allow us to access hardware at such a low level. As a result, we are sometimes required to rely on the operating systems for providing us with timing information. We also must be careful not to rely on any timing mechanism whose performance we are actually evaluating (for example, timing mechanisms within Xinu as guest are something we evaluate in section 9.4).

When running Xinu alone we are able to directly access the hardware clock. As described in section 6, the WRT54GL contains a special count register which is incremented every other processor cycle. Since our MIPS processor runs at 200MHz, this translates to the count register being incremented every 10 nanoseconds. About 42 seconds can elapse before the count register wraps around. Obtaining the value of the count register requires two instruction (mfc0 and a nop), which is negligible overhead. When benchmarking Xinu alone, we read the value of the count register before and after our desired code execution and calculate the difference to determine the elapsed time.

When running OpenWrt alone or Xinu as guest we do not have direct access to the hardware clock. Instead, we rely on the Linux kernel's clock. From user space we use the `gettimeofday` system call. Making a system call has considerably more overhead than reading the count register, but it is the only viable option we have. Executing the system call from within Xinu as guest is equivalent to the hypercall mechanism discussed in section 7.1; for benchmarking OpenWrt alone, the system call is no different from the conventions of a normal user process. We have measured the overhead of the system call to be about 2.318 nanoseconds. To obtain timing information within our module, we use the same clock in the Linux kernel but directly read from the clock structure instead of making a system call.

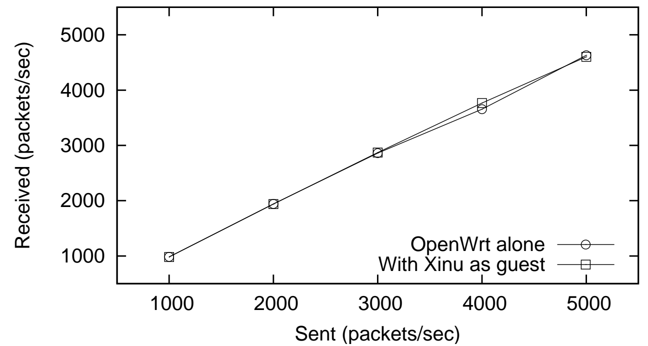

Figure 5: Network throughput

9.2 Processing and Memory Performance

Our first evaluation criteria focuses on the overall performance of the virtual machine. The nqueens problem is used as a processor intensive benchmark. The goal is to place n queens on an n x n chess board such that no queen is able to capture another queen using standard chess moves. We use an iterative solution for boards of size 8 x 8, 10 x 10, and 12 x 12. An array summing algorithm is used as a memory intensive benchmark. The algorithm allocates two one-dimensional arrays of size n and fills them with integer values 0 to n-1. The algorithm then sums the values at each index into a third array of size n. We use arrays with between 10000 and 90000 elements. The benchmarks were run on OpenWrt alone, Xinu alone, and Xinu as guest to compare the performance of the three configurations. The results of the benchmarks are shown in table 1.

9.3 Network Performance

One of the motivations of building a VMM was to isolate new services from the core networking functionality of the router. To meet this goal, the router needs to route packets at almost the same speed as before. We measure the network performance of the router by using two end-hosts to generate and receive traffic. Each end host is equipped with a gigabit Ethernet card and connected to one of the LAN ports on the router. One host generates UDP packets, with a 1000 byte payload of zeroes, at a constant rate; the other end host receives the packets and measures the rate of packets per second. Packets are sent and received for 10 seconds. Figure 5 shows the expected and measured packet rates for OpenWrt alone and OpenWrt with Xinu as guest running the nqueens benchmark.

As the figure shows, there is no measurable difference between the router running OpenWrt alone and running OpenWrt with Xinu as guest. OpenWrt receives and processes network packets exclusively in kernel space, so routing functionality is given precedence over user level processes. This results in Xinu as guest receiving less processing time as the network load increases. Also, it is possible a difference may emerge with higher traffic loads, but the network hardware of the end hosts are unable to send packets reliably at higher

Benchmark	OpenWrt alone (ms)	Xinu alone (ms)	Xinu as guest (ms)
nqueens(8x8)	6.366	5.210	5.279
nqueens(10x10)	159.689	135.067	136.798
nqueens(12x12)	5227.73	4524.27	4584.21
arraysum(10000)	3.447	3.295	3.356
arraysum(20000)	7.257	6.918	7.048
arraysum(30000)	17.355	10.001	10.219
arraysum(40000)	21.993	14.091	14.380
arraysum(50000)	29.205	16.720	17.062
arraysum(60000)	34.317	19.891	20.276
arraysum(70000)	77.704	23.307	23.772
arraysum(80000)	88.802	28.447	28.965
arraysum(90000)	99.479	30.708	31.329

Table 1: Execution time for processing and memory intensive benchmarks

Benchmark	Xinu alone (ms)	Xinu as guest (ms)
Sleep(1000)	999.93	998.164
Sleep(2000)	1999.93	1995.72
Sleep(3000)	2999.93	2993.52
Sleep(4000)	3999.93	3991.37
Sleep(5000)	4999.93	4989.19

Table 2: Observed sleep time

Instruction	Execution Time (μs)
`mtc0 v0, c0_cause`	39.932
`mfc0 v0, c0_status`	40.207
`cache 0x8 0(v0)`	38.884

Table 3: Execution time for privileged instruction emulation

rates.

9.4 Timer Accuracy

Xinu relies on timer interrupts for maintaining the system clock and making appropriate scheduling decisions for processes. To ensure proper functioning of Xinu as guest, we measure the accuracy of the timer using the sleep function. The measurement code stores the value of the system clock, sleeps for a specified number of seconds, and again retrieves the value of the system clock to determine the elapsed time. It is important to note that we rely on the system clock in OpenWrt when measuring timer accuracy in Xinu as guest. This separates the clock we are using as a baseline from the clock whose accuracy we are attempting to measure. Table 2 lists the observed clock times on Xinu alone and Xinu as guest.

Xinu as guest exhibits up to 11 milliseconds of clock skew, while Xinu alone is consistently within a few microseconds of the expected sleep time. Based on the results, it is expected that the clock skew in Xinu as guest will increase as the sleep time increases. This clock skew has multiple potential causes. The first is a queueing of timer interrupt signals. If the VMM does not immediately process the clock interrupt signals, multiple signals may be queued. When the signals are eventually processed, the Xinu interrupt handler is called multiple times in rapid succession, speeding up the clock and causing a shorter sleep duration than expected. Another possible cause is the latency between when an actual interrupt occurs and when Xinu as guest receives the interrupt. We discuss this latency in section 9.6 below.

9.5 Privileged Instruction Handling

As discussed in section 5, the VMM uses a signal handler to emulate privileged instructions. This design makes privileged instructions costly when compared to running Xinu alone. Running Xinu alone, the processor only needs to execute a single instruction; running Xinu as guest, the processor needs to execute hundreds of instructions to dispatch to the signal handler, emulate the instruction, and return control to Xinu. We measure the overhead of emulating instructions by capturing the system time, executing a privileged instruction in Xinu as guest, and capturing the system time again to calculated an elapsed execution time. Table 3 lists the average time to execute a few different privileged instructions.

The overhead of using a signal handler to emulate privileged instructions is a significant amount. The normal execution time for these privileged instructions is a few nanoseconds. Based on the timing measurements, the signal handler is several thousand times slower. However, the use of privileged instructions is limited to a few functions within Xinu – primarily functions related to startup, interrupt control, and interrupt handling. The relative infrequency of privileged instructions makes the overhead more tolerable.

9.6 Timer Interrupt Handling

A signal handler is also used for timer interrupts, in conjunction with a secondary interrupt handler, as discussed in section 6. For reasonable performance we need to: 1) spend minimal time in the secondary interrupt handler, 2) have low latency from interrupt occurrence to guest OS handling of the interrupt, and 3) have interrupts delivered to the guest at accurate intervals. The timer accuracy measurements already evaluate the third criteria.

To evaluate the time spent in the secondary timer interrupt

handler, we measure the time difference between when the handler starts and when it completes. Averaging over the middle 80% of 1000 interrupts, the secondary handler takes 14.823 μs, with a standard deviation of 2.378 μs.

We measure the latency from timer interrupt occurrence to guest OS handling by calculating the time deference between when the secondary interrupt handler starts and when the signal handler completes. Again averaging over the middle 80% of 1000 interrupts, the measured latency is 223.606 μs, with a standard deviation of 1.083 μs. OpenWrt configures the hardware clock to fire a timer interrupt every 4 ms. Using the measured latency as a measure of time spent handling timer interrupts for the VM, about 5.59% of the total available processing time is consumed by this task. Since the measured latency does not include the execution time of the normal timer interrupt handler in OpenWrt, this processing time is largely overhead specific to running Xinu as guest.

9.7 Memory Usage

Processing time is not the only constrained resource in the system. With only 16MB of RAM we are also concerned about the memory usage of our VM. Running the `free` command in OpenWrt before launching the VM reports 13656 KB of available system memory with 8740 KB in use. This implies the OpenWrt code is 2728 KB in size, including 4 KB of reserved space at the start of physical memory. The compiled Xinu code we run in the VM is 140 KB in size. The small size of the Xinu image provides greater flexibility in the amount of memory we need to allocate to the VM. We chose to allocate 1024 KB of memory to the VM: 4 KB reserved memory, 140 KB for Xinu code, and 880 KB heap space. Our vmm-launch process also allocates 4 KB of special memory for storing the registers of our VCPU. Only about 100 bytes of this memory space is used, but we can only allocate memory in 4 KB page sizes. Even while running Xinu as guest, 3888 KB of system memory is still available for OpenWrt to use as necessary. If more services were running in the guest OS, it may be advisable to increase its memory allocation and utilize some of this available memory in the host OS.

9.8 Code Modifications

The final evaluation criteria is the number of code modifications required to OpenWrt and Xinu to successfully run Xinu as guest. Our initial goal was to implement full virtualization with no modifications to Xinu. However, due to time constraints, we opted for paravirtualization with some modifications to the guest OS code.

The code modifications required to Xinu span 12 files, plus the creation of a new device driver for virtualized I/O. Excluding the device driver, we modified about 50 lines of MIPS assembly and about 100 lines of C code. The new device driver is about 150 lines of C code. Our Xinu as guest image is composed of about 11,100 lines of C code and about 500 lines of MIPS assembly in total. Our code changes affect about 2% of the C code and 10% of the assembly code. Further virtualization efforts will allow us to decrease these percentages, with an eventual goal of no modifications.

The majority of code for virtualization resides within the VMM module and vmm-launch process running in OpenWrt. The VMM module consists of about 300 lines of C code. The vmm-launch process – which includes code to prepare the virtual environment and custom signal handlers for interrupts and privileged instructions – consists of about 425 lines of C code. We also made a single one line change to the 2.6.30 Linux kernel in `arch/mips/kernel/cevt-r4k.c` to allow the timer IRQ to be shared amongst multiple interrupt handlers. As we move closer to full virtualization in the future, the code size of the VMM module and vmm-launch will certainly increase.

10. RELATED WORK

System-level virtualization has attracted the attention of the operating system community since the 1970s. It provides many benefits for reliability [9] and cluster computing [14]. Recent virtual machines monitors that have received much attention include VMware ESX Server [13] and Xen [4]. Both of these target the x86 architecture, but present some ideas that are relevant across architectures. VMware uses a technique called ballooning to help cope with overcommitment of memory. Xen uses I/O rings to manage data transfer between devices and guest operating systems.

Virtualization for the MIPS architecture is not as popular, but a few VMMs exist. Disco [8] is the most notable. It is designed to run commodity operating systems on a cc-NUMA multiprocessor computer, for which no dedicated operating system exists. A few "jail" based solutions also exist for MIPS. Linux V-Server [2] and OpenVZ [3] modify the Linux kernel to provide isolated execution environments, but neither is considered to provide either full or paravirtualization.

Virtualization on embedded systems is a relatively new area of interest. The possibility and difficulties in building hypervisors on mobile devices been discussed by [10] and [11]. The main problems in mobile devices described in the papers are battery life and physical-position privacy. Our project does not face either of these issues. MobiVMM [15] is an actual VMM implementation for mobile phones that addresses some of the issues of battery life and embedded device resources constraints.

11. CONCLUSION & FUTURE WORK

We have built a virtual machine monitor to run Embedded Xinu as a guest OS within OpenWrt. The system uses paravirtualization, with some modifications made to Xinu. The VMM is composed of a Linux kernel module and a user-level process. Sufficient support is provided to allow Xinu to startup, handle timer interrupts, provide I/O, and execute processes in the Xinu shell. The virtual machine performance for processor and memory intensive tasks is similar to the execution time when running OpenWrt or Xinu alone. Network throughput is preserved and the virtual machine's clock is relatively accurate. The VMM is currently suitable for wide spread use.

There are two possible directions for future work. One is to decrease the modifications required to Xinu and work towards full virtualization. Achieving this goal requires handling of multiple signal handlers simultaneously. In addition, the VCPU needs to be adapted to provide a virtualized UART device that will work with the normal UART device

driver in Xinu. The second direction for future work is providing support for more components of the Xinu operating system. We disabled networking and memory protection due to time constraints, but a full VMM should support both of these. Network virtualization will require the use of NAT within the host OS to differentiate between traffic destined for OpenWrt and traffic directed to Xinu. Memory protection will require virtualization of the hardware TLB. These improvements will allow Xinu as guest to provide more services and fully satisfy the motivations for the project.

12. REFERENCES

[1] DD-WRT. `http://www.dd-wrt.com`.

[2] Linux-vserver. `http://linux-vserver.org`.

[3] OpenVZ. `http://wiki.openvz.org`.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177. ACM, 2003.

[5] D. Brylow. Embedded xinu project wiki. `http://xinu.mscs.mu.edu`.

[6] D. Brylow. An experimental laboratory environment for teaching embedded hardware systems. In *WCAE '07: Proceedings of the 2007 workshop on Computer architecture education*, pages 44–51, New York, NY, USA, 2007. ACM.

[7] D. Brylow. An experimental laboratory environment for teaching embedded operating systems. *SIGCSE Bull.*, 40(1):192–196, 2008.

[8] E. Bugnion, S. Devine, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 143–156, New York, NY, USA, 1997. ACM.

[9] P. M. Chen and B. D. Noble. When virtual is better than real. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 133, Washington, DC, USA, 2001. IEEE Computer Society.

[10] L. P. Cox and P. M. Chen. Pocket hypervisors: Opportunities and challenges. In *HOTMOBILE '07: Proceedings of the Eighth IEEE Workshop on Mobile Computing Systems and Applications*, pages 46–50, Washington, DC, USA, 2007. IEEE Computer Society.

[11] G. Heiser. The role of virtualization in embedded systems. In *IIES '08: Proceedings of the 1st workshop on Isolation and integration in embedded systems*, pages 11–16, New York, NY, USA, 2008. ACM.

[12] Linksys. WRT54GL wireless-G broadband router. `http://www.linksys.com`.

[13] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[14] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, 2007.

[15] S. Yoo, Y. Liu, C.-H. Hong, C. Yoo, and Y. Zhang. Mobivmm: a virtual machine monitor for mobile phones. In *MobiVirt '08: Proceedings of the First Workshop on Virtualization in Mobile Computing*, pages 1–5, New York, NY, USA, 2008. ACM.