Virtual WiFi: Bring Virtualization from Wired to Wireless

Lei Xia*, Sanjay Kumar§, Xue Yang§
Praveen Gopalakrishnan§, York Liu‡, Sebastian Schoenberg§, Xingang Guo§

* *Northwestern University, Department of Electrical Engineering and Computer Science*
lxia@northwestern.edu

§ *Intel Labs, Hillsboro, OR*
{*sanjay.k.kumar, xue.yang, praveen.gopalakrishnan,sebastian.schoenberg, xingang.guo*}*@intel.com*

‡ *Intel Labs, Santa Clara, CA*
yorkliu@hotmail.com

Abstract

As virtualization trend is moving towards "client virtualization", wireless virtualization remains to be one of the technology gaps that haven't been addressed satisfactorily. Today's approaches are mainly developed for wired network, and are not suitable for virtualizing wireless network interface due to the fundamental differences between wireless and wired LAN devices that we will elaborate in this paper. We propose a wireless LAN virtualization approach named *virtual WiFi* that addresses the technology gap. With our proposed solution, the full wireless LAN functionalities are supported inside virtual machines; each virtual machine can establish its own connection with self-supplied credentials; and multiple separate wireless LAN connections are supported through one physical wireless LAN network interface. We designed and implemented a prototype for our proposed *virtual WiFi* approach, and conducted detailed performance study. Our results show that with conventional virtualization overhead mitigation mechanisms, our proposed approach can support fully functional wireless functions inside VM, and achieve close to native performance of Wireless LAN with moderately increased CPU utilization.

Categories and Subject Descriptors D.4.4 [*Software*]: OPERATING SYSTEMS—Communications Management [Network communication]

Keywords Wireless, Virtualization, WiFi, Hypervisor, Performance

1. Introduction

Virtualization enables multiple operating systems to run simultaneously in isolated containers on a single physical machine and provides an abstraction layer to separate the underlying hardware from what the OS inside a Virtual Machine (VM) observes. Virtualization has already been widely adopted in data centers, where the technology has helped to consolidate servers and dynamically manage existing resources more efficiently.

The technology is moving towards "client virtualization", where a virtual machines run on end users' devices, from notebooks to smart phones, providing unification across a plethora of devices and securing disparate software environments. In particular, enterprise IT has been one of the driving forces behind this technology. The ability to provide strong separation between not only personal and enterprise data but also applications, the operating systems and the entire security configurations/managements, supports the emerging "bring your own device" trend without compromising necessary IT control. Imagine a user brings one's beloved device to the company where IT simply drops a corporate virtual machine onto it. On the device of the user's choice, IT gets the control necessary over all corporate data and applications while users retain the same control for their personal data and applications. The "client virtualization" trend is evidenced by the rapidly emerging desktop/mobile virtualization solutions on the market [16, 36, 37].

As compelling as client virtualization is, the variety of hardware technologies that are predominantly present in client systems make it much more complicated than creating server-based technology. Wireless virtualization represents one of such major technical obstacles. Today's client virtualization solutions typically map all network connections to a virtual wired 802.3 Ethernet card for simplicity. Using a legacy wired adapter works very well for data transfers but has a major downside for wireless connections: features of the underlying network infrastructure or the wireless adapter cannot be controlled from inside the VM and have to be configured and managed at the level of the Virtual Machine Monitor or the hosting OS. While such an approach is reasonable in server virtualization, it doesn't support mobile and ultra-mobile computing very well. In these environments, users have high mobility and often connect exclusively through wireless. They expect sophisticated software tools to manage their network connectivity based on location, availability, performance and cost. For example, only if the connection manager in the VM is aware of changes about wireless link conditions, it could handle the handover between cellular 3G connection and Wireless LAN connection whenever needed.

IT-managed security using IEEE 802.11 together with 802.1x certificate based authentication adds another perspective to consider. A network connection is established only if the certificate is valid and the device is permitted to access the network. The problem with current wireless virtualization approach is that the VMM

VEE'11, March 9–11, 2011, Newport Beach, California, USA.

is the only component aware of the wireless network and performing the authentication with the network. All VM network traffic is then bridged through the single connection, which gives all VMs access to the same network. As a result, maintaining a secured corporate environment has to rely on higher layer solutions such as VPN, which adds cost, connection overhead and increases power consumption.

In this paper, we propose a wireless LAN virtualization approach named *virtual WiFi* that addresses the above two issues. With our solution, the full wireless LAN functionalities are supported inside virtual machines; each virtual machine can establish its own connection with self-supplied credentials. Multiple separate wireless LAN connections are supported through one physical wireless LAN network interface. Our main contributions are as follows:

- We propose a new approach named *virtual WiFi* that is suitable for wireless LAN virtualization.
- We designed and implemented a prototype for our proposed virtual WiFi approach, and performed detailed performance study.
- We identified overhead sources for our approach and we show that with conventional virtualization overhead mitigation mechanisms, our proposed approach can support fully functional wireless functions inside VM, and achieve close to native performance of Wireless LAN with moderately increased CPU utilization.

The rest of the paper is organized as follows: Section 2 discusses the basics of the wireless LAN and contrasts them with wired networks. Section 3 presents our proposed virtual WiFi approach. Section 4 describes our prototype implementation based on the Linux Kernel Virtual Machine (KVM). We discuss the details of our performance study in Section 5 followed by the related work in Section 6. We conclude the paper and present our future work in Section 7.

2. Wireless LAN vs. Wired LAN

Wireless networks have fundamental characteristics that make them significantly different from traditional wired LANs. While a wired LAN device must be physically attached to the wire in order to communicate, wireless devices communicate with each other via electromagnetic wave that has no visible boundaries. Any conformant device with reasonable signal reception can share the air medium and establish its own communication link. In the design of wired LANs, it is implicitly assumed that an address is equivalent to a physical location. On the other hand, in wireless LAN, the addressable unit is a station (STA), where each STA indicates a message destination and is not associated with a particular location.

We describe Wireless LAN in more details below to facilitate deeper understanding of the differences between wireless LANs and wired LANs. IEEE 802.11 defines the wireless LAN (WLAN) standard. Throughout this paper, we use "WiFi", "WLAN" or "wireless LAN" interchangeably. The basic service set (BSS) is the basic building block of an IEEE 802.11 WLAN, where two types of BSS: the infrastructure BSS and the independent BSS (IBSS), are defined. Within an infrastructure BSS, member STAs communicate via a central Access Point, while within an IBSS, two STAs can communicate with each other directly. The widely deployed WLAN systems are primarily infrastructure BSS, which will be the focus of this paper. Figure 1 illustrates a WLAN system that can consist of multiple Basic Service Sets. Each Basic Service Set has a limited coverage since electromagnetic wave signal will be attenuated as it propagates through space, which constraints the range a wireless device can directly communicate with. Multiple BSSs provide extended coverage.

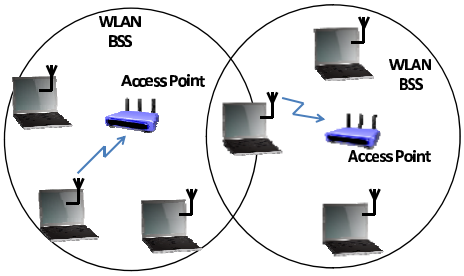


Figure 1. WLAN system with multiple BSSs

WLAN systems primarily operate in two unlicensed radio frequency bands: 2.4 GHz band and 5 GHz band. There are 14 channels designated in the 2.4 GHz band and 42 channels designated in the 5 GHz band[1]. When powered on, a WLAN STA will scan the available channels to discover active networks where Access Points are present. Once the STA has found an Access Point and decided to join its BSS, it will pursue becoming a member of the BSS.

A STA's membership in a BSS is dynamic as the STA turns on, turns off, moves within range, or moves out of range. To become a member of a BSS, a STA needs to authenticate itself, synchronize with the Access Point and become associated with the BSS. IEEE 802.11 Authentication and Association occur at the Access Point prior to any upper layer authentication (e.g., IEEE 802.1X link-layer network authentication). IEEE 802.11 Authentication requires that a STA establishes its identity before sending frames, where Open System Authentication uses the STA's MAC address as authentication identify, and Shared Key authentication supports several different shared key authentication mechanisms.

Once authentication has completed, stations can associate with an AP to access all services of the BSS. Association is logically analogous to plugging a cable into a wired network and it allows the AP to record each STA so that frames may be properly delivered. Only after the association process is completed, a station is capable of exchanging data frames with the access point.

A defining characteristic of the wireless channel is the variation of the channel condition over time and over frequency. In addition, a wireless device can experience significant interference from various sources. As a result, complex management functions have been used for WLAN radios in order to achieve efficient and reliable communication. Such management functions often involve the device driver[2] for control and configuration, which makes them important to consider when designing a solution for WiFi virtualization. Examples of such management functions include:

- Rate adaptation: Data streams at WLAN device are modulated and transmitted over the air at a certain rate. IEEE 802.11 mandates multiple transmission rates (e.g., 802.11g supports twelve rates from 1 to 54 Mbps). Higher data rates are commonly achieved by more efficient modulation schemes, which typically require a stronger signal to decode. In wireless networks, path loss, fading, and interference often cause variations in the received signal quality, which in turn, cause variations in the accuracy of decoding given a modulation scheme. A trade-off

[1] Depending on the regulations, the specific channels allowed in different countries may vary.

[2] The WLAN MAC software is typically divided into two entities: μCode run by a real-time micro controller on the WiFi device and a device driver that is part of the host operating system.

generally emerges: the higher the data rate, the higher the probability of receiving errors. Rate adaptation is the process of dynamically switching data rates to match the channel conditions with the goal of selecting the rate that will give the optimum throughput under the present channel conditions.

- Power management: Mobile devices with WLAN radio usually have a limited energy budget constrained by battery life. On the other hand, the shared channel access nature of IEEE 802.11 forces wireless STAs to continuously listen to the channel to determine its current status. As a result, a mobile device using WLAN radio would drain its battery very quickly. IEEE 802.11 standard provides power save modes to reduce the time required for a station to listen to the channel. The device driver can control how long and how often the radio needs to be on.
- Power control: Transmit power of WLAN devices affects many aspects of the underlying wireless network. It determines the range of a transmission, the quality of the signal received at the receiver as well as the magnitude of interference it causes to other receivers. The typical goal of power control is to set the transmit power of a WLAN device to the lowest possible level that is still compatible with the quality of the desired communication.

The above brief introduction of Wireless LAN provides a glimpse into the complexity of the Wireless LAN device, especially on the complexity of WLAN management functions. Compared with wired LAN devices that involve mainly data centric operations, their differences primarily lie in the following aspects:

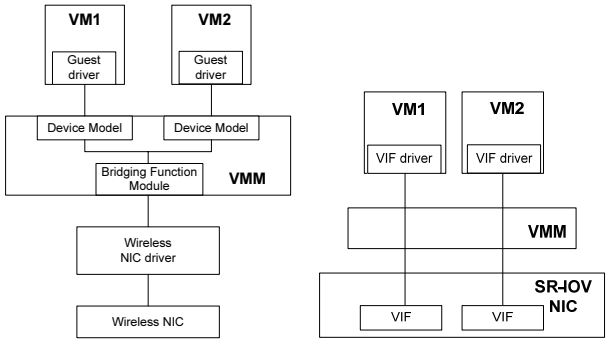
1. There is a range of complex management functions that affect the fundamental functionalities of wireless LAN devices. The device driver is generally involved in many of those management decisions for the WLAN device to have acceptable performance. On the contrary, wired LAN devices are data centric and have very little management functions.
2. Wireless STA is not constrained by the physical location or the number of available network plugs. Inherently, multiple wireless links can be setup from one mobile device without the Access Point knowing that multiple wireless links originate from the same device.
3. Wireless LAN throughput is not bounded by the platform's I/O bandwidth. Rather, it is bounded by the wireless channel capacity. Additionally, due to the distributed channel access and dynamic nature of the wireless link, the channel utilization ratio is typically less than 50%. The achievable throughput to date is generally less than 20 Mbps for 802.11a/g (peak channel rate is 54 Mbps) and less than 200 Mbps for 802.11n (peak channel rate is 450 Mbps) [15].

3. Wireless LAN Virtualization

3.1 Limitations of current virtualization approaches

Today's network interface virtualization techniques can be categorized as either software or hardware based approaches. The software based approach is shown in Figure 2(a), where the VMM implements the virtualization functions in software [10, 35] to support virtual network interfaces for multiple guest VMs. The VMM establishes the actual network connection using the platform's physical NIC and then bridges the connection to multiple virtual machines. In many implementations, the selected virtual network card is a legacy ethernet card for simplicity, and the guest OS inside the VM uses the standard off-the-shelf ethernet device driver. In some other implementations, the para-virtualized driver will be used in the guest OS for function/performance enhancements.

The second approach focuses on providing hardware virtualization support on the NIC device itself. In particular, Single Root I/O Virtualization (SR-IOV) [29] provides a standard mechanism for devices to advertise their ability to be simultaneously shared among multiple virtual machines, and it allows for the partitioning of a PCI function into a set of virtual interfaces. As shown in Figure 2b, a SR-IOV enabled NIC device presents multiple virtual interfaces (VIF) and the VMM can directly assign a VIF to a specific virtual machine, hence drastically reducing the performance penalty of high-bandwidth network cards such as 10Gbit Ethernet.



(a) Software-based approach (b) Hardware-based approach

Figure 2. Existing network interface virtualization approaches

Existing approaches are primarily developed for wired network interface, and are not suitable for virtualizing wireless network interface due to the fundamental differences between wireless LAN and wired LAN devices we elaborated in Section 2. More specifically, the limitations of the 802.3-based emulation approach come from its difficulty to support Wireless LAN management functions inside the VM. IEEE 802.11 is required to appear to higher layers (logical link control (LLC)) as a wired IEEE 802 LAN. This requires IEEE 802.11 to incorporate functionality that is untraditional for MAC sublayers, in order to meet reliability assumptions and to handle QoS traffic in a manner comparable to wired LANs. In other words, IEEE 802.11 MAC functions is a super set of 802.3 MAC functions, and many management functions will get lost when emulating IEEE 802.11 device as 802.3 device.

Using a para-virtualization approach and have the VMM vendor supply the para-virtualized driver is technically possible. However, given the WLAN management functions are complex and the management interface between host driver and wireless LAN device is often proprietary, the reality is that VMM vendor would only provide the smallest common denominator of many wireless network cards. Any vendor specific feature or software component that provided additional benefits would not be possible inside the guest.

Hardware based virtualization approach provides virtualization support at NIC device, where the limitations mainly lie in the cost and complexity – validation and consolidation of management commands from multiple VMs have to be implemented on the device. Due to the wide range of management functions a WLAN device has to support, hardware based virtualization mechanism such as SR-IOV would significantly increase the complexity and cost of wireless NIC. As we mentioned before, wireless LAN throughput is not bounded by I/O bandwidth, given the best achievable wireless LAN throughput is less than 200 Mbps to date. As such, hardware based virtualization mechanisms do not add any additional value in boosting the data I/O bandwidth.

3.2 virtual WiFi: Proposed Wireless LAN Virtualization Architecture

In this paper, we propose a wireless LAN virtualization architecture shown in Figure 3 that takes a combined approach of software and hardware. The goal is to support native wireless LAN drivers inside guest VMs and allow each VM to establish its own wireless connection, all with minimum wireless LAN device changes. The desire to pass on the benefits of the various functions provided in a particular wireless network card leads to the approach of exposing to the guest VM the very same network card as the physical WiFi card. This is contrary to the desire to using Virtualization for a full abstraction of the underlying hardware. However, a similar approach has been taken with SR-IOV and we argue that such an approach is more suitable than the full device abstraction for wireless devices given their management-intensive nature.

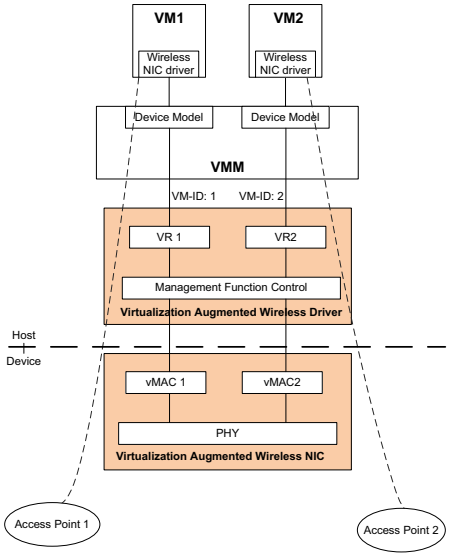


Figure 3. Proposed architecture for virtualizing WLAN network interface

We introduce the concept of *assisted driver direct execution* (ADDE), where an augmented wireless LAN host driver—residing either in a driver domain of a type I hypervisor or in the Host OS in case of a type II hypervisor—takes the primary role in supporting the management functions of the virtualized wireless interfaces. At the same time, some commands (e.g., TX command for data transmissions) will be directly passed on to wireless LAN device for execution. Since the virtual Wireless LAN devices exposed to the guest VMs are the same as the physical wireless LAN device, the commands issued by the guest VM drivers are understandable by the physical device. The augmented host driver can forward those commands to wireless LAN device without incurring any translation overhead.

Inside the virtualization augmented wireless NIC, virtual Wireless LAN interfaces are separated at MAC layer. As shown in Figure 3, multiple virtual MACs may be active and they share the common wireless physical layer via time domain multiplexing. As we mentioned previously, wireless STA is not constrained by the physical location. When multiple virtual MACs are running, each of them can initiate the authentication/association procedures, and setup its connection with an Access Point. From the view of the Access Point, it would not know that those wireless LAN connections are associated with virtual Wireless LAN interfaces located on the same device.

Given that the device-specific virtual wireless functions are provided by the augmented wireless driver and the wireless NIC, the VMM itself now only needs to emulate standard PCI config and MMIO space functions in its device model to expose a wireless network card into the guest. Such an architecture choice enables wireless vendors to provide their own wireless virtualization solutions and to show value differentiations, with minimum dependence on VMM software. Additionally, such an architecture choice allows the device model owned by VMM vendors to be agnostic to changes of the driver-μCode interfaces owned by WiFi device vendors.

Since the device model exposes the same device as the physical wireless interface to the VM, the guest OS inside the VM will load the card's native driver. Once the device model receives commands from the guest's driver, it will tag the commands with a *VM-ID* specific for each guest VM. Functional commands issued by a VM device driver will be forwarded by the device model to the augmented host driver. The device model is also responsible for injecting interrupts into the guest if needed.

3.3 Virtualization Augmented Wireless LAN Device

A typical Wireless LAN device consists of the RF transceiver that performs RF signal transmitting/receiving; the baseband section that operates at a lower frequency range and performs mainly digital signal processing; and the MAC portion that deals with link establishment, security, channel access mechanism, etc. Baseband and RF sections are generally referred to as physical layer (PHY) and the MAC layer runs on top of the PHY. The MAC portion often consists of a real-time controller on the device with associated μcode software, as well as the driver running on the host computer. As we mentioned previously, virtual wireless LAN interfaces are separated at the MAC layer.

The concept of having two MAC entities sharing the same PHY to function as two wireless interfaces in a non-virtualized environment has emerged in the wireless industry recently. It is driven by usage cases of supporting WiFi peer-to-peer connections while connecting to an infrastructure access point at the same time using a single WiFi radio. There, one MAC entity operates in the standard client wireless LAN (STA) mode to connect to an access point for Internet access, while the other operates as a software access point to manage a wireless personal area network (streaming video to a TV equipped with WiFi using the peer-to-peer connection, for example). Commercially available products include Intel My WiFi Technology [20], Atheros Direct Connect[9], and Marvell Mobile Hotspot [17]. Building upon such technology feasibility, we further expand it to support wireless virtualization in virtualized environments, where multiple MAC entities will operate in STA mode and will maintain their independent associations with corresponding access points.

The reason that multiple MAC entities sharing the same PHY can function as separate wireless interfaces is due to the broadcast nature of the wireless link. As long as the RF transceiver is tuned to a particular channel, it can transmit/receive packets on that channel. In the case where all virtual MACs operate on the same channel, the RF transceiver just need to stay on that channel and receive all packets. Irrelevant packets can be filtered out using MAC filters to save power and computing cycles. It is MAC layer's responsibility to identify a packet based on the BSS ID and source/destination MAC addresses. If different virtual MACs need to operate on different channels, then careful time-multiplexing scheduling across virtual MACs needs to be done to make sure each vMAC maintains synchronization with the corresponding access point, and it listens to the channel at the right time for intended traffic. Detailed scheduling algorithms specifically for multi-channel opera-

tions concern specific details of IEEE 802.11, which is out of scope of this paper.

Each virtual MAC will have a separate MAC address. The μCode on the network card will maintain information/state per connection associated with a VM-ID, which includes the access point MAC address, STA MAC address, data rates supported by the access point, operating channel, security credentials, rate scaling table, etc. More specifically, on the receiving path, with properly configured MAC filters, packets associated with all virtual MACs can reach the MAC layer. The μCode then determines the corresponding VM-ID based on destination MAC address, and program the hardware with appropriate parameters such as security method in use and corresponding security key to decrypt the received packet. The decrypted packet is then tagged with the appropriate VM-ID, and sent to the augmented host driver, which will in turn, route the packet to the corresponding VM.

On the transmission path, the augmented driver communicates directly with the card's μCode. Handling of the data transmission command is straightforward and the augmented driver passes the TX command that was tagged with the VM-ID by the device model directly to the card. The μCode then looks up the information/state associated with this VM-ID and applies the corresponding security key to encrypt the packet in case of an encrypted connection. The encrypted packet will then be transmitted over the air with the modulation rate, power and antenna configurations associated with this connection. Handling of management commands is more involving. As guest drivers from multiple VMs can issue independent commands, they can potentially conflict with each other (e.g., one VM likes to turn off the wireless LAN device while other VMs are still using it). To resolve such a conflict, μCode maintains per-VM data structures to keep VM specific states isolated from each other. A guest driver can modify its own state, but any command that affects the operation of the entire wireless NIC is carefully handled at the augmented driver to ensure consistent isolation among VMs. Several commonly used management commands are elaborated below.

Device initialization. Initialization command is used to bring up the PHY and set up all parameters needed for an association, such as band selection, channel selection, transceiver chain configurations, security method, BSS ID, etc. Each VM WiFi driver may independently issue device initialization commands. If one VM is actively using the WiFi device while another VM issues device initialization command, μCode needs to make sure that PHY setting of the active connection will not be affected. Additionally, the device initialization command should trigger μCode to start a new vMAC, and start maintaining state/information related to the new vMAC.

Scan request. Scan request command is issued by WiFi driver to request the WiFi device to search for all available channels to discover active networks where access points are present. Each VM WiFi driver can independently issue scan request commands. If one VM is actively using the WiFi device while another VM issues scan request, the augmented host driver needs to consolidate function requests from both VMs and set proper scan request for the μCode to execute. More specifically, the scan request is characterized by how often the WiFi device needs to switch to an unknown channel to search for available access points (i.e., Scan Interval), and how long it will stay on that channel for each scan attempt (i.e., Scan Duration). Depending on traffic amount of the active VM, the augmented host driver may need to set the Scan Interval and Scan Duration differently from what is requested by the guest VM. Alternatively, the augmented driver can also return the previously stored scan results directly to the VM without passing scan command to μCode, if the available scanning results were just recently obtained.

Power save command. Power save request is issued by WiFi driver to control how long and how often the WiFi radio needs to be on to save power consumption while maintaining the active connection. It sets sleep interval to allow the WiFi radio to go into low power mode. When different VMs issue power save commands, the augmented driver needs to consolidate these requests and determine a sleep schedule that satisfies the connection and traffic requirements of all VMs. Only one consolidated power save command will be passed down to the μCode for execution.

Rate control command. The WiFi driver can adaptively choose a set of suitable rates for the WiFi radio to be used, based on packet transmission/failure histories and based on channel feedbacks from the μCode. If a guest VM issues the rate control command, such a command should be passed down to μCode, where the μCode should only update the rate table associated with the specific VM-ID.

TX power control command. WiFi driver can specify TX power that should be used for a particular channel or modulation rate. When a guest VM issues the TX power control command, such a command should be passed down to μCode, where μCode should only update the TX power table associated with the specific VM-ID.

In some cases, μCode on the WiFi device needs to send management feedback up to the WiFi driver. Such management feedback may be delayed responses to a command that was originally issued by a VM, or it may be statistical information on packet error rate, the number of retransmissions performed for each packet, etc. The augmented host driver is responsible for routing those management feedbacks to the appropriate VM.

4. Prototype Implementation

We have implemented the *virtual WiFi* prototype in KVM (Kernel-based Virtual Machine). The core part of KVM [3, 21] is a Linux module that enables the Linux OS to function as a VMM. In KVM, VMs run as normal Linux processes. Since all physical pages of a VM are mapped into the process's user virtual memory space, the physical memory of the VM can be accessed easily when the VM process is in user mode. KVM takes advantages of hardware virtualization support such as Intel VT [6] or AMD-SVM [8] to achieve efficient virtualization of CPUs and memory.

The KVM can specify on which I/O ports the requests from VM need to be intercepted. Whenever an IO request to these ports happens in VM, a VM exit happens and processor switches control from guest to KVM. In KVM, the I/O request is redirected to a user mode device emulator called QEMU [11, 12], which is utilized by KVM to provide virtual device models to VMs.

Figure 4 shows the overall architecture of our virtual WiFi prototype implementation, which consists of three main components. The first is the *virtual WiFi device model*, which is implemented inside QEMU device model layer, responsible for exposing the virtual WiFi interface to the VMs. The second one is the *Virtualization augmented WiFi device driver* running in the host Linux kernel. The third one is the *Virtualization augmented WiFi NIC*, which involves only the μCode changes on the WiFi device. We will elaborate each of the three components in the virtual WiFi prototyping system in the following subsections.

4.1 Virtual WiFi Device Model

The prototype implements an Intel WiFi device model which works for both Intel's 5000 and 6000 series WiFi cards. Each VMM that supports virtualization of WiFi device through virtual WiFi approach will implement its own version of the virtual WiFi device model. The virtual WiFi device model virtualizes the PCI config and MMIO accesses itself, while forwarding all WiFi function commands to the augmented driver for processing. All IO requests

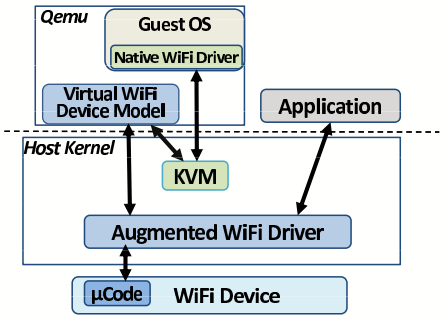


Figure 4. Virtual WiFi Prototype implementation based on KVM

from guest WiFi driver are intercepted by KVM and delivered to the virtual WiFi device model, which then hands these requests to the augmented host WiFi driver.

During initialization virtual WiFi device model establishes an ioctl interface to the augmented host driver, and uses this interface to allocate a new VM state inside the driver. For all commands/packets sent by guest WiFi driver, the device model tags them with host driver allocated VM-ID and deliver them to the augmented host driver via the ioctl interface. For received packet or incoming interrupt from physical WiFi NIC, the augmented host driver signals the device model about the availability of a packet or an interrupt. The device model retrieves the received packets/interrupts via an ioctl call. If the received packet is valid, device model enqueues it in the guest WiFi driver queue and injects an interrupt into the guest OS for further processing.

4.2 Virtualization Augmented Host Driver

The virtualization extension is added to the default "Intel Wireless WiFi" driver in Linux kernel release 2.6.33 [4] that works for both Intel's 5000 and 6000 series WiFi cards. The extension implements an IOCTL interface to interact with device model inside QEMU. On the transmission path, when the augmented driver receives a command from the device model, it either virtualizes the command locally, or validates the command and inserts it into the physical WiFi driver transmission queue (data or control queue). On the receiving path, when receiving a packet from wireless device, it examines the VM-ID associated with the packet to identify the intended receiver. It then signals the corresponding device model which wakes up from a waiting poll to pick up the packet.

4.3 Virtualization Augmented WiFi NIC

The μCode handles time critical MAC operations. The virtualization extension is added to WiFi μCode of both Intel's 5000 and 6000 series WiFi cards. Both Intel's 5000 and 6000 series WiFi cards have Intel My WiFi technology enabled, so the NIC hardware is capable of handling at least two virtual MACs. Configuration, connection status and state machine are maintained separately for each virtual MAC. Control/data messages to/from each vMAC will be tagged with different VM-ID so that they can be differentiated by the augmented host driver.

μCode changes made in our prototype are as follows. When a new virtual machine is initiated, a mapping table is created to map VM-ID with the virtual MAC, and configure the hardware filtering policy to allow packets targeted to this virtual MAC to be received. On the receiving path, VM-ID is identified based on the received packet's MAC addresses and the packet is tagged with the VM-ID before sending it to the host driver. On the TX path, command response is generated after completing the command execution. The command response is also tagged with VM-ID before sending it to the host driver. The security keys are maintained in a unified security table, which is indexed by the combination of connecting access point and the virtual MAC address.

4.4 Address Translation

Typical device models copy TX packets from the VM's memory and send them to host networking stack for transmission. However, since virtual WiFi device model has a direct interface to host driver, it can avoid the extra copy for transmission packets. Once virtual WiFi device model receives a TX command from the VM that contains the guest physical address (GPA) of the packet buffer, the device model requests VMM to convert the GPA to host physical address (HPA) instead of copying the packet from the VM's memory. After the GPA in the TX command being replaced with the HPA, the TX command is sent to the wireless NIC by the host driver for actual transmission. The wireless NIC performs the DMA operation using the HPA, copying the packet directly from the VM's memory to avoid extra memory copy.

It was observed that address translation by the VMM in software causes significant CPU overhead. Thus one optimization is to exploit the address remapping hardware support present in the platform, such as Intel VT-d [6] or AMD IOMMU [8]. We implemented both software-based and hardware-assisted address translation in our virtual WiFi prototype for performance analysis. The hardware-assisted address translation is based on Intel VT-d in our implementation. VT-d hardware is typically used to support assignment of a single device to only one address domain (and hence only one VM). That is, only one VT-d context entry can be assigned for each PCI device. The GPAs used by a device in DMA requests are mapped to HPAs through the VT-d remapping table pointed by the device's VT-d context entry. On the other hand, in our virtual WiFi system, one physical wireless NIC has to be shared by multiple VMs, which will require one remapping table for each VM that shares the physical device. Virtual WiFi implements an approach that enables VT-d to support multiple address domains within a single VT-d context entry, by merging multiple VT-d tables into a single VT-d table.

The approach we implemented is based on the fact that current VT-d table has much larger physical address space than the physical memory available to each VM. Therefore a number of high order bits in GPAs will remain un-programmed in typical usages. We can partition the VT-d table into multiple chunks in a way that each chunk contains mappings for one VM. For example, if the VT-d hardware supports mapping up to 512 GB memory address space (39 bits long, 3-level page table), we can divide the address space into 16 chunks with each chunk supporting up to 32GB address space. In this way, VMM creates a merged VT-d table where each chunk represents the unique remapping sub-table for each VM.

During a VM's initialization, the virtual WiFi device model requests the VMM to add the VM's VT-d table into the merged VT-d table used by the device. The VMM selects the chunk location corresponding to the VM-ID and copies the top-level entries from the VM's table into the merged table. Using the same example as above, 32 top-level entries will be copied from the VM table to the merged VT-d table at chunk location 1 if the VM-ID is 1. When the virtual WiFi device model receives commands from guest driver, the device model integrates the VM-ID into the higher order bits (bits 35–38 in this example) of the GPA present in the command. When the command is sent to the wireless NIC, the device uses the VM-ID tagged GPA for DMA request, which will lead to GPA being translated to HPA using the correct chunk corresponding to the VM in the merged VT-d table.

5. Performance Evaluation

The goal of our performance analysis is to understand whether our proposed virtual WiFi architecture is an applicable approach for wireless virtualization. More specifically, we would like to understand, in comparison with native WiFi, is there any WiFi throughput or latency penalty; what is the CPU utilization by wireless virtualization; what are the major overhead sources for WiFi virtualization and what are the effective mitigation mechanisms.

To measure the UDP goodput, we used *IxChariot* [2] network benchmarking tool. The IxChariot tools contain both Console and Performance Endpoint programs. Performance Endpoints are installed at the source and destination points of a test to perform packet transmissions and receptions. The Console program configures the test, loads the test scripts to Endpoints and collects/processes test results. In our test setup, the Performance Endpoints run on the two test machines and the Console program runs on a separate management machine. The separation of management console from test machine allows us to isolate the CPU usage by Console program from the CPU usage due to WiFi traffic generated by the EndPoint [3].

To isolate the test management traffic (e.g., loading test scripts or gathering test results) from the testing traffic, the management network and the test network are separated into two different subnets. The primary test machine on which VM and virtual WiFi will be running connects with an access point via WiFi. The other end of the tested data connection is a machine running native Linux and connects to the access point via Ethernet. As such, a packet from the primary test machine will go through the WiFi link to reach the access point, which will then route the packet to the peer test node through Ethernet. As the WiFi link is the bottleneck link, the testing throughput reflects the throughput supported by the WiFi.

In our evaluation, Chariot 7.0 *UDP-Throughput* script is used to measure the end to end UDP goodput between the primary test machine and the peer test node. End to end TCP throughput is measured by *iPerf* [1]. To measure latency, we used the round-trip delay of an ICMP echo request/response pair (i.e., ping), by taking samples over hour-long intervals. All reported results are averaged over multiple measurements.

The primary test machine is a HP EliteBook 6930p laptop with Intel Core2 Duo CPU at 2.53GHz, 4 GB RAM, 80 GB HDD, and Intel WiFi 5300 AGN card. Both the host OS and guest OS are 32-bit Linux with kernel version 2.6.33.1. For the performance results below, we disabled one of the two CPU cores. The access point used is Cisco WAP410N. For our purpose of analysis, we performed the tests in several different scenarios:

- *Native:* Tests are performed in Linux with native WiFi driver. No virtual machine is running in this case. The original WiFi μCode is loaded onto the device. This is the baseline of the native WiFi performance. We refer to results in this scenario as *Native*.
- VM-Passthrough: Tests are performed in a Linux Guest OS, with the native WiFi driver running inside the guest VM. The VM is configured with 2G RAM and a single core CPU. The physical WiFi device is assigned directly to the VM with Intel VT-d support. In this case, the guest WiFi driver interacted directly with the physical device. The original WiFi μCode is loaded onto the device. Neither device model nor augmented host driver is involved. We use this configuration to capture the performance when there is hardware supported WiFi virtualization. We refer to the results in this configuration as *Passthrough*.
- Virtual WiFi: Tests are performed in a Linux Guest VM with the native WiFi driver running inside the guest VM. The augmented host driver runs in host Linux kernel, and the augmented μCode is loaded onto the device. The VM is configured with 2G RAM and a single core CPU. The virtual WiFi interface is exposed by virtual WiFi device model. This configuration follows our proposed approach as shown in Figure 4. The software-based address translation approach is applied unless VT-d is specifically mentioned. We refer to results in this configuration as *Virtual WiFi*.

To analyze virtual WiFi overhead contributing factors, we used the Oprofile [5] tool to analyze the performance behaviors on different components of the system. Oprofile can attain a system-wide statistical profiling results for Linux-based systems, which include codes executing at both user and kernel levels. In our profiling, we mainly focus on the CPU cycles (*UNHALTED_CPU_CYCLES* event) consumed by each of the major components of virtual WiFi prototype system. OProfile 0.9.6 is used in the tests. The profiling result is presented and analyzed in section 5.3.

5.1 Throughput

Throughput test results are shown in Figure 5, where TX/RX throughput is measured when the primary test machine transmits/receives to/from the peer test node respectively. UDP throughput is shown in Figure 5(a), where Chariot are configured to use 1450 byte packet size with traffic enough to saturate the WiFi link. Each test lasts for 60 seconds. The measured throughput has large variation across different runs due to the WiFi channel variation. The presented throughput is averaged over ten runs. As we can see, virtual WiFi achieves throughput comparable to the native case. In all testing scenarios, achieved WiFi UDP TX/RX throughput is around 22 Mbps/18 Mbps respectively using IEEE 802.11g WiFi mode. Similar observations can be made for TCP throughput tested using iPerf, as shown in Figure 5(b). In all testing scenarios, achieved WiFi TCP TX/RX throughput is around 20 Mbps/20 Mbps respectively using IEEE 802.11g WiFi mode. In the tests, iPerf is configured to use 1448 byte writes, the socket buffer size is maximized, and 4 million writes are made.

Both UDP and TCP throughput tests confirm that our proposed WiFi virtualization approach achieves throughput comparable to native WiFi case.

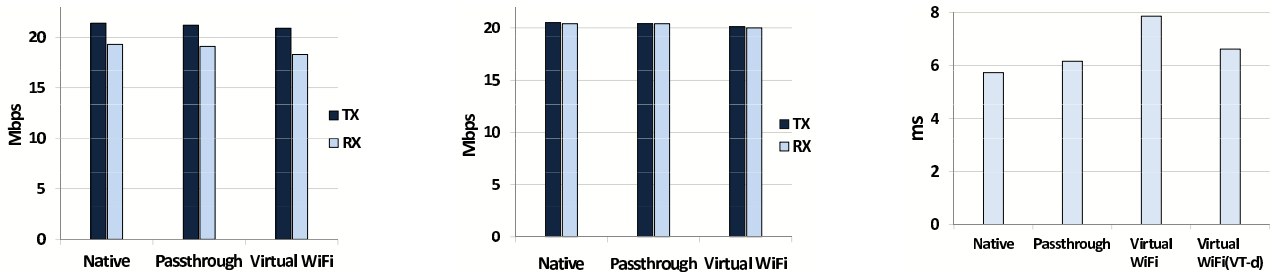
5.2 Latency

Latency test results are shown in Figure 5(c), where round-trip latency is measured when the primary test machine pings the peer test node. The latencies are the average of 15 measurements. The results show that the latency of virtual WiFi system is about 35% higher than the native case (7.87 ms vs. 5.74 ms). In the case of VM-Passthrough, the latency is 7% higher than native case (6.17 ms vs. 5.74 ms).

To understand where this extra latency comes from, we break down the latency of transmission path into various components. More specifically, we define the *virtualization extra path* as the additional path in the TX/RX critical path introduced by virtualization. In the virtual WiFi prototype system, the virtualization extra path on network transmit starts from the point when guest driver issues a packet sending (MMIO) request until the point the requests (packets) are sent to the TX queue of the physical WiFi device by the augmented host driver.

Our first instrumentation gauges the time spent on transmitting a packet by reading the processor's *Time Stamp Counter* (TSC) register at key points during the virtualization extra path on network transmit. The TSC allows a measurement of the total cycle count of the path and breakdown of interesting subsegments.

[3] Performance Endpoint itself is an unobtrusive software agent with light CPU load.



(a) UDP Throughput (b) TCP Throughput (c) Round-trip Latency

Figure 5. End-to-end Performance of Virtual WiFi (Throughputs and Latency)

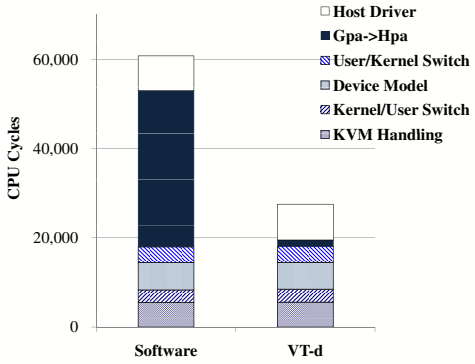


Figure 6. Cycle number spent on major components on virtualization extra path for single packet sending

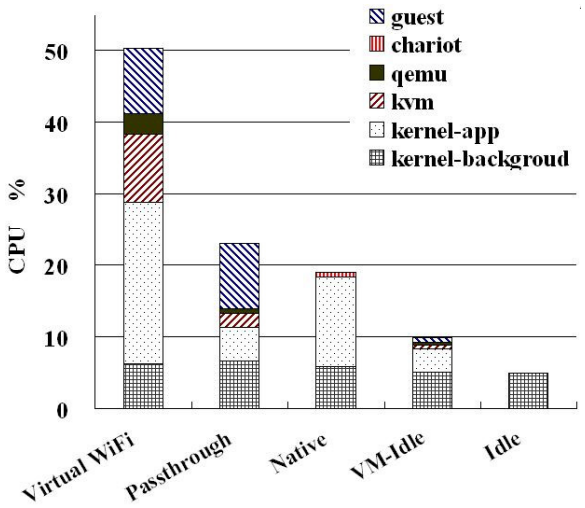


Figure 7. Overall CPU usage breakdown

Figure 6 presents the transmission latency break down involved along the instrumented virtualization extra path using virtual WiFi with software-based address translation mechanism and hardware translation support (VT-d). It takes a total of 60000 CPU cycles from the start of MMIO instruction from guest until packet is put in the TX queue of the physical WiFi device for software based address translation approach. The address translation (marked as "GPA → HPA" in Figure 6) in software-based approach is responsible for almost half of these cycles for network transmit.

With hardware IOMMU supports, we can reduce the latency caused by software address translation. Specifically, in our prototype system, we eliminated most of the latency caused from address translation by enabling VT-d support, and the round trip latency of virtual WiFi with VT-d is only 15% higher than the native case (6.63 ms vs. 5.74 ms in Figure 5(c)).

5.3 CPU Usage

Given that virtual WiFi achieves close to native WiFi throughput and latency, we are interested to see how much system resource is consumed by virtual WiFi. In particular, mobile devices such as smartphones and tablets typically have limited processing power, which may lead to CPU being a bottleneck. In this section, we dissect the CPU time consumed by each of the major components of virtual WiFi prototype system when the test machine is performing network transmission in maximal throughput. The goal is to understand what are the major overhead sources in terms of CPU consumption, as well as to identify effective mitigation mechanisms to reduce such overhead.

5.3.1 Overall System CPU Usage

Figure 7 shows the overall system CPU usage measured at the primary test machine in different test scenarios. In each non-idle test scenario, the primary test machine is performing UDP transmission in its maximal throughput. As we can see, when in system idle state without virtual environment running, the CPU usage is 5% due to background processes and Oprofile daemon itself. With one VM running idly, the system CPU usage is 10%. When Chariot test workload is running in host Linux and transmitting over native WiFi without any virtualization, the system CPU usage is 19.5%. When running Chariot in guest VM with passthrough WiFi device assigned, the system CPU usage is 23.6%. Finally, when Chariot is running in Linux guest VM using our proposed virtual WiFi approach without VT-d being enabled, the system CPU usage is 50.9%. Clearly, virtual WiFi consumes much more CPU time than the native WiFi case. In comparison, the CPU time consumed in VM-Passthrough case is much less.

To investigate the sources of overheads, we used Oprofile to determine where the CPU time is spent. Oprofile uses time-based sampling to profile the distribution of time spent in main components over the entire workload. The samples measure the percentage of time spent in code sections and the number of samples that hit a section. This gives a more comprehensive picture of the overheads present in transmitting packets and reveals some expensive functional components.

The CPU time consumption is broken down into percent of time spent in the major components of virtual WiFi system including:

- ***Guest***: The guest VCPU thread, running guest OS and its applications.
- ***Chariot***: The test workload. In native case, Chariot's CPU time is shown separately. However, in virtual WiFi and passthrough cases, Chariot is running in guest, thus its CPU time is included in the guest as OProfile cannot separate applications running inside VM.
- ***QEMU***: The QEMU user process, which contains mainly the virtual WiFi device model.
- ***KVM***: The KVM module running inside host kernel, this part is separated from the host Linux kernel for more analysis. However, some of KVM code calls routines in other part of kernel, the time spent by those routines is accounted to the host kernel.
- ***Kernel-App***: The part of host Linux kernel time that runs on behalf of processing user requests, which includes host augmented driver and all kernel service for QEMU, KVM and other functions to support WiFi virtualization.
- ***Kernel-Background***: The background routines running in kernel, including Oprofile daemon itself. This part consumes almost the same for all test scenarios, which we treat it as background CPU time unrelated to virtualization.

The overall CPU usage and distribution of CPU time on these major components described above is presented in Figure 7. We can observe that most of the extra CPU time are consumed by host kernel and KVM for virtual WiFi comparing with passthrough and native cases. It should be noted that in the passthrough case, the network stack is running inside Guest OS instead of Host OS, which is the reason why Figure 7 shows *Kernel-App* in passthrough case consumes less CPU time than the native case.

5.3.2 Host Kernel and KVM CPU Time

Table 1 further break down the CPU time distributed in KVM module and host Linux kernel (*Kernel-App* part in Figure 7), as we observed they are responsible for most of the extra CPU time consumption when running virtual WiFi. By comparing the CPU time distribution on host kernel and KVM for virtual WiFi with passthrough and native cases, we made the following observations:

Interrupt Handling. The WiFi driver in Linux kernel release 2.6.33 used in our prototype implementation sets interrupt coalescing timer to 2 ms [4]. However, it is observed that the μCode actually overrides the interrupt coalescing timer set by the driver and enables immediate interrupt for each TX command response. As a result, each transmitted packet results in a TX command response interrupt. This explains why in native WiFi case, the system CPU usage is relatively high (19.5%). In the case of virtual WiFi, when a physical interrupt arrived at the augmented host driver, the host driver will notify the device model running in QEMU, which in turn injects a virtual interrupt to the guest through KVM. As a result, one physical interrupt could trigger four components' involvements in series, which includes four times of kernel/user and VMM/VM switch. It obviously leads to much more CPU consumption if the interrupts happen in high frequency.

Address translation. In measures done for Figure 7 and Table 1, virtual WiFi uses the software-based address translation approach. Most of address translation work is done by KVM module, and it is another significant source of overhead. By applying hardware IOMMU support, this part of overhead is supposed to be eliminated or significantly reduced.

IO handling. Whenever guest driver issue a MMIO access, the control transits into KVM by VM exit. KVM is responsible for either handling the requests or forwarding it to the device models in QEMU. After the device model is done with MMIO handling, it returns to KVM, which resumes the guest VM. KVM IO handling takes a significant chunk of CPU time also. The I/O handled in KVM also includes accesses to the virtual interrupt controller. IO handling is also a major CPU cost in the host kernel. The I/O handling cost in host kernel corresponds to two parts: the primary one is due to handling *ioctl* system calls from the virtual WiFi device model and the other part is the I/O read/write to the physical wireless NIC by the augmented host driver.

Contention/Synchronization. In virtual WiFi case, threads synchronization/contention/scheduling cost is one of the major CPU overhead in host kernel. Recall that in KVM/QEMU architecture, guest VM run as separate threads from the device models. Any interactions between VMs and device models, such as MMIO accesses by guest OS to virtual devices, or virtual interrupts injected by device models into VMs, result in synchronization between VM threads and device model threads. The large CPU time spent in handling thread synchronization in host kernel indicates there are a large number of interactions between VMs and device models. Such interactions include MMIO accesses from guest and interrupt delivery/injection to guest, which further confirms our observations that I/O handling and interrupt handling are the major sources of overhead for our virtual WiFi prototype system.

Comparing with our virtual WiFi approach, the direct passthrough approach obviously needs much less kernel involvements. This is because every IO request from the guest VM is passed directly to physical device, and physical interrupt from physical device is delivered to guest by host kernel and KVM without the involvement of either host driver or device model.

5.3.3 Overhead Mitigation Mechanisms

The CPU time distribution breakdown results from above discussions indicate that interrupt handling and address translation consume large portions of CPU time in our prototype implementation, which can be mitigated using interrupt coalescing and the hardware-assisted address translation mechanism (e.g., VT-d), respectively.

Interrupt Coalescing The interrupt handling in the particular test case we analyzed mainly comes from TX command responses. After a TX command is sent to the WiFi device, a command response will be sent back by the device after the transmission attempt is completed. The command response can be sent to the augmented host driver by the device at various times through interrupt. The augmented host driver will then notify the device model running in QEMU, which then injects a virtual interrupt to the guest. Obviously, interrupts have a higher impact in such a virtualized environment due to additional involvements of the device model in QEMU and the augmented host driver. Most network devices today support *interrupt coalescing*, which is to delay generating the interrupt until either a specified number of packets is received or when the maximum specified delay is reached. The WiFi driver in Linux kernel release 2.6.33 used in our prototype implementation sets interrupt coalescing timer to 2 ms [4]. However, it is observed that the μCode actually overrides the interrupt coalescing timer set by the driver and enables immediate interrupt for TX command response. As a result, each transmitted packet results in a TX command response interrupt. With the average throughput of 22 Mbps and the payload size of 1450 bytes, we are observing close to 1900 interrupts per second. Without the interrupt coalescing at WiFi device, we implemented similar function at the device model, where the device model wait for 10 TX responses before it injects a virtual interrupt to the guest, in order to observe the mitigation effect

KVM Time		
Category	Percent of Total Time	
	Virtual WiFi	Passthrough
Delivering virtual IRQs	2.79%	0.34%
Address Translation	2.71%	0.15%
IN/OUTs handling and forwarding	2.06%	0.13%
Instruction Decoding	0.53%	0.33%
Managing guest shadow memory	0.69%	0.36%
Virtual CPU state updating	0.64%	0.44%

Host Kernel Time			
Category	Percent of Total Time		
	Virtual WiFi	Passthrough	Native
Interrupt handling/forwarding IRQs to device model	5.87%	1.22%	2.53%
IN/OUTs in driver/Handle IO requests from device model	4.95%	0.29%	2.56%
Locking/unlocking code section	4.72%	0.41%	1.08%
Scheduling user/kernel threads	2.06%	0.69%	0.30%
Packet memory copying	1.74%	0.35%	1.57%
Timer management/Timing service	1.15%	0.71%	0.10%
System call entry/return	1.78%	0.68%	0.56%
Other	0.50%	0.20%	0.34%
Network Stack			3.47%

Table 1. Distribution of CPU time spent in KVM and host kernel.

of interrupt coalescing. As we can see in Figure 8, with interrupt coalescing at the device model, CPU utilization is dropped from 50.4% to 34.2% while achieving the same throughput. We expect even less CPU utilization if the interrupt coalescing is enabled at the WiFi device, since the interrupt coalescing at the device will limit the number of physical and virtual interrupts, while the interrupt coalescing we implemented at the device model only limits the number of virtual interrupts.

Hardware-assisted Address Translation After enabling VT-d, we observed the CPU utilization is dropped from 50.4% to 44.2% as shown in Figure 8. With both interrupt coalescing and VT-d are enabled, the total CPU utilization is 28% while achieving the same throughput. Recall that the total CPU utilization is 19.3% in the native WiFi case with no virtual machine running.

The reduction of IO handling overhead cannot be achieved as long as the device model and host driver are involved in each guest I/O request. One of our future works is to further investigate the impact of *data passthrough* on wireless virtualization. More specifically, we are investigating an approach that can support data traffic passthrough between different guests and the physical device through separate queues. However, the management traffic will be intercepted by the hypervisor and forwarded to the augmented host driver for further handling. Such an approach can take advantage of the flexibility of software (i.e., augmented host driver) to support complex WiFi management functions. On the other hand, it has the potential of removing extra virtualization overhead from the critical path of data TX/RX.

In summary, with interrupt coalescing and hardware-assisted address translation, virtual WiFi achieves throughput comparable to the native WiFi case with relatively 45% more CPU utilization (28% vs. 19.3% CPU utilization).

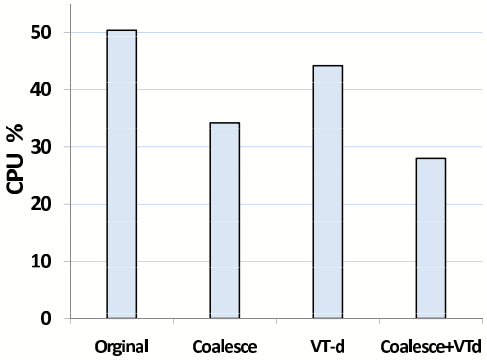


Figure 8. CPU usage with different optimizations

6. Related Work

Virtualization The concept of virtualization has been widely studied in the context of operating system. Various optimizations have been proposed to improve the virtual machine monitors such as Xen, VMware, Palacios [10, 23, 38, 39]. In recent years, there has been a fast growing interest in mobile/client virtualization [22]. Examples of client virtualization solutions on the market include XenClient [16], VMware desktop virtualization solutions [37] and VirtualLogix mobile virtualization solutions [36]. Existing research on I/O virtualization can be categorized into three approaches: Fully emulated I/O, para-virtualized I/O, and hardware-assisted virtualized I/O.

Full emulated I/O virtualization [35] provides virtualized views of devices by emulating real devices instead of providing physical devices. Device models (virtual devices) implement virtualized hardware devices completely in software within the VMM. Multiple virtual devices are then be multiplexed on a single physical device. In this approach, no guest software change is required. But there is significant performance overhead, due to large number of context switches between VM and hypervisor caused by guest's IO accesses. Para-virtualization I/O [10, 25, 27, 31] refers to modifying guest OS to run on the virtualized environment and exposing some details of the hardware for optimization. Xen I/O extends this concept by requiring guest changes and having the special driver

talk to a special VM that has direct hardware access. Drivers can also be placed into separate driver VMs for better protection [24]. These techniques can often lead to significantly better I/O performance. Hardware-assisted virtualized I/O that exploits specialized hardware [7, 26, 29, 30, 32] has been proposed as an efficient I/O virtualization mechanism. With hardware assisted virtualization, the device presents multiple logical interfaces which can be securely accessed directly by guest OS. Close to native I/O performance can be achieved as a result of bypassing the hypervisor. However, hardware virtualization requires substantial investments in re-designing the device. In addition, hardware-assisted virtualized I/O breaks the device transparency, which is a benefit of using software-based device virtualization. Device transparency has the benefit of avoiding the need to maintain device-specific code in guest VMs, also simplifying live migration of guest VMs across physical machines that have different flavors of devices.

Our proposed virtual WiFi approach is similar to emulated I/O approach in that it supports unmodified WiFi device driver in guest OS. However, it is different from emulated I/O in that it requires device model emulate the same device as the physical device on the host and it passes many of the commands to the physical device without emulating it. Therefore, it can achieve better performance than emulated I/O approach. On the other hand, different from hardware-assisted virtualized I/O, virtual WiFi approach does not depend on any specific I/O virtualization support on device. Given WiFi devices available on the market[9, 17, 20] can already support more than one vMACs, only software (i.e., driver and μCode) upgrade is needed for supporting virtual WiFi solution. Essentially, virtual WiFi is a hybrid I/O virtualization approach that is more suitable for the purposes of exposing full wireless functionalities inside guest VM, providing WiFi link level isolation, simplifying hardware and software design and achieving appropriate performance/complexity tradeoff.

There also have been many existing works that study the performance overhead of different virtualization approaches on different hypervisors and platforms [14, 18, 19, 28, 32]. This paper analyzes the performance overhead of our proposed wireless virtualization approach. To our best knowledge, we are not aware of any existing work that addresses wireless virtualization and conducts associated performance study.

Wireless Network/Radio Virtualization Sachs *et al.* [33] proposed a radio resource sharing framework, which allow different virtual radio networks to operate on top of a common shared infrastructure and share the same radio resources without interfering with each other. Smith *et al.* [34] designed and implemented a virtualized 802.11 testbed system with the goal to allow multiple experiments to co-exist on a wireless experimental facility. Their key issues are to maintain the coherence of each wireless experiment (i.e., transmitter/receiver/potential interferers all need to operate on the same channel), and to keep different experiments isolated (i.e., one experiment will not affect results of another experiment). Miljanic *et al.* discussed virtual radio in the context of software defined radio. Their goal is to provide a common interface to layers above the link layer so that the radio resource scheduling/allocation is hidden from higher layers. A radio virtualization layer is introduced to handle the hardware resources responsible for managing communication bandwidth. MultiNet [13] proposes a driver based approach to facilitate simultaneous connections to multiple networks using a single wireless card. The "virtualization" of wireless card is implemented with intermediate driver, which continuously switches the card across multiple network. Microsoft Windows 7 has adopted this feature, which is not designed to support virtual machine environment.

7. Conclusion and Future Work

We proposed a hybrid wireless virtualization approach named *virtual WiFi*, which combines elements of software and hardware for virtualization support. The design goals of *virtual WiFi* are to expose full wireless functionalities inside guest VM, to provide WiFi link level isolation across different VMs, and to achieve appropriate performance/complexity tradeoff with architecture choices specifically suitable for wireless interface virtualization.

We have designed and implemented a prototype supporting the IEEE 802.11g compliant WiFi interfaces. Work is ongoing to extend the prototype to support IEEE 802.11n compliant WiFi interfaces that operate at higher speeds. We are also porting our virtual WiFi implementation to Xen and plan to understand virtual WiFi performance in the context of Xen.

We showed that with interrupt coalescing and hardware-assisted address translation, *virtual WiFi* achieves throughput comparable to the native WiFi case with relatively 45% more CPU utilization (28% vs. 19.3% CPU utilization). We also plan to further investigate the impact of *data passthrough* on wireless virtualization as we mentioned in Section 5.3.3.

References

[1] iperf homepage. `http://iperf.sourceforge.net/`.

[2] Ixchariot homepage. `http://www.ixchariot.com/`.

[3] Kvm homepage. `http://www.linux-kvm.org/`.

[4] Linux kernel 2.6.33, iwlwifi driver. `http://lxr.linux.no/linux+v2.6.33/drivers/net/wireless/iwlwifi/`.

[5] Oprofile homepage. `http://oprofile.sourceforge.net/`.

[6] ABRAMSON, D., JACKSON, J., MUTHRASANALLUR, S., NEIGER, G., REGNIER, G., SANKARAN, R., SCHOINAS, I., UHLIG, R., VEMBU, B., AND WIEGERT, J. Intel virtualization technology for directed I/O. *Intel Technology Journal 10*, 3 (2006), 179–192.

[7] ANWER, M. B., AND FEAMSTER, N. Building a fast, virtualized data plane with programmable hardware. In *ACM SIGCOMM Workshop on Virtualized Infastructure Systems and Architectures* (2009).

[8] APIKI, S. I/O Virtualization and AMD's IOMMU. Advanced Micro Devices, Inc. `http://developer.amd.com/documentation/articles/pages/892006101.aspx`.

[9] ATHEROS COMMUNICATIONS, INC. Direct connection. `http://www.atheros.com`.

[10] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)* (2003), pp. 164–177.

[11] BARTHOLOMEW, D. Qemu: a multihost, multitarget emulator. *Linux J. 2006*, 145 (2006), 3.

[12] BELLARD, F. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), USENIX Association, pp. 41–41.

[13] CHANDRA, R., BAHL, P., AND BAHL, P. Multinet: Connecting to multiple ieee 802.11 networks using a single wireless card. In *IEEE INFOCOM, Hong Kong* (2004).

[14] CHERKASOVA, L., CHERKASOVA, L., GARDNER, R., AND GARDNER, R. Measuring cpu overhead for I/O processing in the xen virtual machine monitor. In *USENIX Annual Technical Conference* (2005), pp. 387–390.

[15] CISCO SYSTEMS, INC. Enterprise wireless competitive performance test results. White Paper, 2010. `http://www.cisco.com/`.

[16] CITRIX SYSTEMS, INC. Xenclient virtual desktops. `http://www.citrix.com/`.

[17] GIORDANO, B. Transforming small mobile devices into full-featured wifi access points. Marvell Semiconductor, December 2009.

http://www.marvell.com/technologies/wireless/marvell_wifi_mobile_hotspo%t_whitepaper.pdf.

[18] GUO, D., LIAO, G., AND BHUYAN, L. N. Performance characterization and cache-aware core scheduling in a virtualized multi-core server under 10gbe. *IEEE Workload Characterization Symposium 0* (2009), 168–177.

[19] GUO, D., LIAO, G., BHUYAN, L. N., LIU, B., AND DING, J. J. A scalable multithreaded l7-filter design for multi-core servers. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (New York, NY, USA, 2008), ANCS '08, ACM, pp. 60–68.

[20] INTEL CORP. Intel My WiFi Technology Tech Brief. http://www.intel.com/network/connectivity/products/wireless/mywifi.htm.

[21] KIVITY, A. kvm: the linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium* (2007), pp. 225–230.

[22] LAMBERT, N. Demystifying client virtualization. Forrester Research, Inc., April 2008. http://www.vmware.com/files/pdf/analysts/Forrester_Demystifying-Client-%Virtualization.pdf.

[23] LANGE, J., PEDRETTI, K., HUDSON, T., DINDA, P., CUI, Z., XIA, L., BRIDGES, P., GOCKE, A., JACONETTE, S., LEVENHAGEN, M., AND BRIGHTWELL, R. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)* (2010).

[24] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GTZ, S. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (2004), pp. 17–30.

[25] LIAO, G., GUO, D., BHUYAN, L., AND KING, S. R. Software techniques to improve virtualized i/o performance on multi-core systems. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (New York, NY, USA, 2008), ANCS '08, ACM, pp. 161–170.

[26] LIU, J., HUANG, W., ABALI, B., AND PANDA, D. High performance vmm-bypass I/O in virtual machines. In *Proceedings of the USENIX Annual Technical Conference* (May 2006).

[27] MENON, A., COX, A. L., AND ZWAENEPOEL, W. Optimizing network virtualization in xen. In *Proceedings of the USENIX Annual Technical Conference* (2006), pp. 15–28.

[28] MENON, A., JOHN JANAKIRAMAN, G., SANTOS, J. R., AND ZWAENEPOEL, W. Diagnosing performance overheads in the xen virtual machine environment. In *VEE '05: Proc. 1st ACM/USENIX International Conference on Virtual Execution Environments* (2005), ACM Press, pp. 13–23.

[29] PCI-SIG. I/O virtualization. http://www.pcisig.com/specifications/iov/.

[30] RAJ, H., AND SCHWAN, K. High performance and scalable I/O virtualization via self-virtualized devices. In *Proc. of HPDC* (2007), pp. 179–188.

[31] RAM, K. K., SANTOS, J. R., TURNER, Y., COX, A. L., AND RIXNER, S. Achieving 10 gb/s using safe and transparent network interface virtualization. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (New York, NY, USA, 2009), ACM, pp. 61–70.

[32] RENATO, J., YOSHIO, S., JOHN, T. G., AND PRATT, J. I. Bridging the gap between software and hardware techniques for I/O virtualization. In *2008 USENIX Annual Technical Conference* (2008).

[33] SACHS, J., AND BAUCKE, S. Virtua radio – a framework for configurable radio networks. In *WICON '08: Proceedings of the Fourth International Wireless Internet Conference* (Maui, Hawaii, USA, November 2008), ACM.

[34] SMITH, G., CHATURVEDI, A., MISHRA, A., AND BANERJEE, S. Wireless virtualization on commodity 802.11 hardware. In *WiNTECH '07* (Montreal, Quebec, Canada, Sept 2007), ACM.

[35] SUGERMAN, J., VENKITACHALAN, G., AND LIM, B.-H. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference* (June 2001).

[36] VIRTUALLOGIX INC. Virtuallogix vlx. http://www.virtuallogix.com/.

[37] VMWARE, INC. Vmware desktop virtualization products. http://www.vmware.com/products/desktop_virtualization.html.

[38] WALDSPURGER, C. A. Memory resource management in vmware esx server. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation* (New York, NY, USA, 2002), ACM, pp. 181–194.

[39] XIA, L., LANGE, J., DINDA, P., AND BAE, C. Investigating virtual passthrough I/O on commodity devices. *SIGOPS Oper. Syst. Rev. 43*, 3 (2009), 83–94.