NORTHWESTERN
UNIVERSITY

Electrical Engineering and Computer Science Department

Technical Report
NWU-EECS-08-11
November 30, 2008

An Introduction to the Palacios Virtual Machine Monitor---Version 1.0

Jack Lange **Peter Dinda**

Abstract

Palacios is the first virtual machine monitor (VMM) from the V3VEE Project to be released for public use as community resource. Palacios is a "type-I", non-paravirtualized VMM that makes extensive use of the virtualization extensions in modern x86 processors, such as AMD SVM and Intel VT. It consists of ~30 KLOC of C and assembly written at Northwestern University. This document describes the overall structure of Palacios and how it works. It also explains how to download Palacios, develop within it, and extend it

This project is made possible by support from the National Science Foundation (NSF) via grants CNS-0709168, CNS-0707365, and the Department of Energy (DOE) via a subcontract from Oak Ridge National Laboratory on grant DEAC05-00OR22725. Sandia National Laboratory has also provided assistance. Jack Lange was partially supported by a Symantec Research Labs Fellowship

Keywords: virtual machine monitors, hypervisors, operating systems, resource virtualization

An Introduction to the Palacios Virtual Machine Monitor—Version 1.0

Jack Lange Peter Dinda

November 23, 2008

Abstract

Palacios is the first virtual machine monitor (VMM) from the V3VEE Project to be released for public use as community resource. Palacios is a "type-I", non-paravirtualized VMM that makes extensive use of the virtualization extensions in modern x86 processors, such as AMD SVM and Intel VT. It consists of $\sim$ 30 KLOC of C and assembly written at Northwestern University. This document describes the overall structure of Palacios and how it works. It also explains how to download Palacios, develop within it, and extend it.[1]

[1]This project is made possible by support from the National Science Foundation (NSF) via grants CNS-0709168, CNS-0707365, and the Department of Energy (DOE) via a subcontract from Oak Ridge National Laboratory on grant DE-AC05-00OR22725. Sandia National Laboratory has also provided assistance. Jack Lange was partially supported by a Symantec Research Labs Fellowship.

Contents

1 Introduction

The V3VEE project (v3vee.org) is creating a virtual machine monitor framework for modern architectures (those with hardware virtualization support) that will permit the compile-time creation of VMMs with different structures, including those optimized for computer architecture research and use in high performance computing. V3VEE is a collaborative project with the University of New Mexico.

Palacios[2] is the first virtual machine monitor under development within the project, and the first to be accessible to the public. Palacios currently targets the IA32[3] architecture (hosts and guests) and makes extensive, and non-optional use of the AMD SVM [1] extensions (partial support for Intel VT [5, 10] is also implemented). It uses shadow paging in the virtual MMU model (nested paging is optional). It runs directly on the hardware and provides a non-paravirtualized interface to the guest. An extensive infrastructure for hooking of guest physical memory addresses (with byte address granularity), guest I/O ports, and interrupts facilitates experimentation. At the present time, Palacios is capable of booting an unmodified Linux distribution. It consists of $\sim$ 30 KLOC of C and assembly written from scratch at Northwestern.

The purpose of this document is to explain the overall structure and operation of Palacios, and to act as a "getting started" guide for those who would like to use, or contribute to, the Palacios codebase.

Palacios is released under a BSD license. The complete license, and the licenses for related tools and components is included in the source code release.

2 What you will need

To run the current version of Palacios, you will need either a physical machine that supports the AMD SVM extensions[4], or an emulator that does so. Palacios currently has incomplete support for the Intel VT virtualization extensions. The specific machines we use in our testbed are:

- QEMU 0.9.1. QEMU [2] can emulate x86 processors on other processors, and does so particularly fast on other x86 processors. This version of QEMU adds support for AMD SVM (without nested paging). Because it is an emulator, you do not actually have to have a physical machine with AMD SVM (or indeed, even an AMD or x86 processor) to test Palacios. We use QEMU extensively for development of Palacios. QEMU runs on Windows as well as most Unix platforms and Linux.
- HP Proliant ML115 with an AMD Opteron 1210 processor, 1 GB RAM, 160 GB SATA drive, and ATAPI CD ROM. The 1210 is among the cheapest AMD processors that support SVM without nested paging.
- Dell PowerEdge SC1435 with an AMD Opteron 2350 processor, 2 GB RAM, 160 GB SATA drive, and ATAPI CD ROM. The 2350 is among the cheapest AMD processors that support SVM with nested paging.
- Dell PowerEdge SC440 with an Intel Xeon 3040 processor, 2 GB RAM, 160 GB SATA drive, and ATAPI CD ROM. The 3040 supports the Intel VT extensions. We use machines like this for development of VT support in Palacios.

[2] Palacios, TX is the "Shrimp Capital of Texas". The Palacios VMM is quite small, and, of course, the V3VEE Project is a small research project in the giant virtualization space.

[3] 32 and 64 bit hosts are supported for running 32 bit guests. Support for 64 bit guests is in progress.

[4] The initial implementations of AMD SVM did not support nested paging. Palacios does not require nested paging support, and so can run on the earliest (and cheapest) Opterons that included SVM.

When using physical hardware, we find it very convenient to use the PXE boot environment. PXE is BIOS-level support for booting the machine over the network. You can easily set up a server machine that provides the Palacios image via PXE. Combined with a network power switch or RSA card, and an IP-KVM, PXE boot support makes it very straightforward to remotely use development and testing hardware.

The software environment needed to build Palacios includes numerous open source or free software tools. Note, however, that we distribute a tool bundle that builds these for you. We also include QEMU so that you can immediately run Palacios, even if you don't have test hardware. The tools include:

- GCC and binutils. To our knowledge, Palacios is compatible with all recent versions of GCC and binutils, however some non-Palacios components we use are not. We include a compiler toolchain sufficient to compile Palacios and all its dependencies.
- nasm and ndisasm. These are needed to compile GeekOS. We include appropriate versions.
- The bcc, as86, and ld86 components of Dev86. These tools are used to build 16-bit components, such as the guest's BIOS. We include appropriate versions of these tools.
- Git is used for version control. We include a complete git install in the VM release.
- Perl 5.8. Various build scripts are written in Perl. We do not include Perl.

If you plan on accessing our public repository, you will need git, or a client compatible with git. If you plan to do testing on a physical machine, you may find kermit to be helpful. Palacios uses host OS output facilities. Most host OSes, including the ones we support, such as GeekOS, send debugging output to the first serial port ("COM1").

3 How to get the code

Palacios is available from the V3VEE Project's web site (v3vee.org). It is available in the following forms:

- **Public git repository:** Git is a version control system originally developed for use with the Linux kernel. Our public repository makes it possible to get access to the latest public version of Palacios, as well as to contribute to its development. You can access the repository both with standard git tools, and using a web browser.
- **Source releases and snapshots:** You can download source releases and snapshots from our web site.
- **Palacios development VM image:** You can download a VMware-compatible[5] image. While this image is large, it contains not only the full Palacios source distribution, but also all the tools needed to build and run it (under QEMU). Once the Palacios development VM is booted (mere seconds), you are immediately ready to go. Since the Palacios code in the VM is checked out from the repository, you can also immediately update to the latest version, and contribute.

Palacios includes or leverages code from the following open source or free software projects.

- GeekOS [4]: GeekOS "Project 0" is used as the default OS in which Palacios is embedded. We use GeekOS to bootstrap the underlying physical machine and provide drivers for the keyboard, serial port, and text screen. We also include a network card driver we have developed.

[5] VMware tools that can run this image, such as VMware Player, are available for free from vmware.com.

- Kitten [9]: Palacios can also be embedded into Sandia National Lab's Kitten operating system. As Kitten has not yet been released, we only include the Palacios-side code to do this.
- BOCHS [6]: Modified versions of the BOCHS ROM BIOS and VGA BIOS are included for bootstrapping the guest VM.
- XED [7]: Intel's XED x86 instruction decoder/encoder from the Pin Project is used to decode instructions in some circumstances.
- LWIP/UIP [3]: The lightweight TCP/IP stack is optionally included to add networking to Palacios.
- VM86 [8]: IBM's virtual 8086 mode emulator is optionally included to provide guest bootstrap support on Intel VT machines. It is unused on AMD machines.
- Ubuntu: The Palacios development VM image is based on Ubuntu 8.10, Desktop Workstation Edition.

The relevant code is included with Palacios—nothing else needs to be downloaded.

3.1 Getting and installing the development VM

Begin by downloading the Palacios development VM image from v3vee.org. The image is a zip file and can easily be unzipped on Linux or Windows. Next, if you have not already done so, visit vmware.com and download and install VMware Player, VMware Workstation, or VMware Server. Be sure to configure support for at least NAT networking.

You should now be able to double-click on the VM startup file (palacios-distro-vm.vmx). You will see a typical Ubuntu boot process. At the login screen, log in as v3vee with password v3vee. In ∼v3vee/palacios, you will find the Palacios source code, as checked out from git. All the tools you need, including QEMU to run Palacios right from the command line, are installed and on your path. Most of the tools are installed in /vmm-tools.

The Palacios Development VM image is configured with NAT networking. This makes it possible for a user of the VM to access the network as a client, but it is not possible to connect to the VM as a server (for example, to "ssh in"). If you need this, change the configuration to use bridged networking.

3.2 Getting and installing a source tarball

Begin by downloading the Palacios release tarball from v3vee.org. Untar it into your working directory. This will create a subdirectory `palacios`. You will now need to install the included development tools. To do this:

```
cd palacios
./SETUP_DEV_ENV.pl
```

This will build appropriate versions of the needed development tools. It will not install an appropriate version of QEMU, which you will need to install separately in order to run Palacios on the development computer.

3.3 Using the git repository

The first thing you need to do is install git on your local system. The download is located at http://git.or.cz. We highly recommend building from source and installing in the default locations.

Once git is working, you can need to configure your personal identification. To do this, run:

```
git config --global user.name "Your Name"
git config --global user.email "your_email_address"
```

Now you can clone from our release repository:

```
git clone http://www.v3vee.org/~jarusl/palacios.git
git checkout --track -b release-1.0 origin/release-1.0
```

The above commands will checkout the head of the release-1.0 branch of Palacios into the subdirectory `palacios`.

Over time, it is likely that we will make available other releases and branches. We will include the appropriate commands (similar to the above) on our web site, or privately.

Git provides for distributed development. By running the above commands, you have created a local repository for your own work in Palacios, based on our release 1.0 branch. You can manage that repository as you will, including giving others access to it. You can also incorporate changes from our repository. Finally, git can package up your changes so you can send them back to us for possible inclusion in our repository.

Although we cannot go into depth on the operation of git here, there are a few commands that you will find useful:

- `git branch` shows the current branch you are working on.
- `git commit` writes your changes to your local clone of the Palacios repository.
- `git pull` fetches changes from our repository and applies them to your checked out branch. (This is like a cvs/svn update)
- `git checkout <branch-name>` changes to another development branch.
- `git stash` is helpful for trying out newly committed changes without having them conflict with your presently uncommitted changes.
- `gitk` is a very useful graphical user interface to git.

If you'd like to learn more about git, we recommend reading the git manual which is available at the above URL. Git makes it very easy to both branch and merge, unlike CVS/SVN.

4 How to compile the code

To compile the code, do the following:

```
cd palacios/build
make world
```

The compilation process does the following:

- It builds the `vm_kernel`, which is the basic software loaded into a guest at boot time. This presently includes a BIOS and a VGA BIOS.
- It builds Palacios as a static library.
- It builds a minimal version of GeekOS.
- It links Palacios, GeekOS, the `vm_kernel`, and a simple boot loader into a multiboot-compliant binary that can be translated into a floppy disk or CD ROM image. These images can in turn be used on physical hardware or via PXE for network booting.

At this point, the Palacios image (`palacios/build/vmm.img`) is ready to be used. However, the guest OS must be provided externally. For example, vmm.img can be written to a floppy, and the test machine booted from that floppy with the guest in the CD ROM drive. Another option is to PXE-boot the test machine over the network from vmm.img with the guest provided in the physical CD ROM.

The guest OS can also be included as the multiboot image RAM disk. Palacios includes a virtualized IDE bus to which the RAM disk can be attached. From the perspective of the guest OS, the RAM disk appears to be a simple CD ROM device. One common use of this functionality is to integrate the Palacios VMM and a guest OS on a single CD ROM image. To do this, execute:

```
cd palacios/build
cp <guest-os-iso-file> palacios/build/iso/puppy.iso
make geekos-iso
```

This will result in the file `test.iso` which is a CD ROM image that contains the combination.[6]

5 How to run the code

We assume that you have built `palacios/build/test.iso` according to the instructions in the previous section. Given that, you can run Palacios and have it boot the integrated guest ISO simply by

```
cd palacios/build
./RunIso.sh
```

You will see the following in succession:

1. The QEMU boot process, including its BIOS
2. The GeekOS boot process
3. The Palacios boot process
4. The boot of the guest, starting with its BIOS, and leading through its normal boot sequence.

[6]The use of puppy.iso in the above is not accidental. We have already configured the process to use vmm.img and puppy.img, as this constitutes our typical test build. If you would like to use different names, you will want to edit menu.lst. You can use any ISO file as the guest OS without doing so.

When the guest is started, it has full control over the screen. Palacios dumps debugging output via host OS facilities. In GeekOS, this is via the first serial port. When run under QEMU using RunIso.sh, this serial output appears in the file `serial.out`. The amount and nature of Palacios debugging information that appears depends on compile-time options, all of which have the prefix `DEBUG`.

The simplest way to run on a physical machine is to write test.iso to an actual physical CD ROM, and then boot the machine from that CD ROM. Generally, testing is much faster using QEMU. Furthermore, although Palacios and the modified BOCHS BIOS we use in the guest generally support ATAPI CD ROMs correctly on physical hardware, it's been our experience that there is considerable variation in that hardware. We have generally found that if there is a problem in booting on a physical machine, tweaking the BOCHS BIOS is needed, which can be painful.

5.1 Debugging with QEMU

It is possible to do Palacios debugging using GDB and QEMU. We do this by starting QEMU with these additional flags:

```
-monitor stdio -S
```

Once you get a monitor prompt, run the following:

```
(qemu)  gdbserver
Waiting gdb connection on port '1234'
```

At this point, you can switch to a different window and fire up GDB with the ELF kernel image:

```
# gdb palacios/geekos/build/geekos/kernel.exe
```

Then set the target to the QEMU port:

```
(gdb) target remote localhost:1234
Remote debugging using localhost:1234 0x000083c0
```

You can now set a breakpoint, for example:

```
(gdb) break RunVMM
Breakpoint 1, RunVMM (bootInfo=0x2fec) at ../src/geekos/vm.c:156
(gdb) c
Continuing.
```

The GDB continue command will start the boot process.

6 Code structure

We now consider the directory structure used by Palacios and then describe the nature of the codebase more deeply.

6.1 Directory structure

The overall directory structure of Palacios is shown below:

```
palacios/
   build/     top-level build directory
   devtools/  created by SETUP_DEV_ENV.pl
   geekos/    default GeekOS host OS
                source, includes, build
   kitten/    Kitten host OS - not in initial distro
                source, includes, build
   [other host OSes here]
   misc/      test code
   palacios/  Palacios codebase
                source, includes, build
                vm_kernel code here as well
   utils/     Utility software
```

At the top level, you will also find copies of the Palacios license and other basic documentation.

You can find more information about GeekOS and Kitten on their respective web sites. The changes needed to support Palacios are generally rather minor. Here is a summary of the most salient GeekOS changes made to support Palacios.

- palacios/geekos/src/geekos/{vm.c, vmm_stubs.c} implement the basic interface with Palacios.
- palacios/geekos/src/geekos/{serial.c} implements serial debugging output from GeekOS.
- palacios/geekos/src/geekos/{ne2k.c, rtl8139.c} implement GeekOS drivers for simple network cards.
- palacios/geekos/src/{lwip,uip} contain the LWIP and UIP TCP/IP stacks.
- palacios/geekos/src/geekos/{net.c,socket.c} implements a simple socket interface for the LWIP/UIP network stack in GeekOS.
- palacios/geekos/src/geekos/pci.c implements basic PCI scan and enumerate functionality for GeekOS.
- palacios/geekos/src/geekos/{queue.c, ring_buffer.c} implement these data structures.
- Corresponding include files are in palacios/geekos/include.
- Other smaller changes are made and can be easily found with a diff against the GeekOS "Project 0" code.

The directory structure of the Palacios codebase (palacios/palacios) is as below:

```
build/              main build directory
                      libv3vee.a lands here
  devices/          virtual device object code lands here
  palacios/         core palacios object code lands here
  xed/              XED interface object code
```

```
  vm_kernel           guest BIOS, etc, lands here
  payload_layout.txt  defines guest layout
  rombios             symlink to BIOS to use in guest
  vgabios             symlink to VGA BIOS to use in guest

include/              main include directory
  devices/            virtual devices
  palacios/           core palacios
  xed/                XED interface

lib/                  libraries to link with palacios
  xed/                XED static library here

scripts/              scripts used to build/manage palacios

src/                  main source directory
  devices/            virtual devices
  geekos/             palacios-side geekos interface code
  kitten/             palacios-side kitten interface code
  palacios/           core palacios code
  vmboot/             basic guest code
    rombios/          guest BIOS (modified BOCHS)
    vgabios/          guest VGABIOS (modified BOCHS)
    vmxassist/        guest v8086 monitor (from IBM rHype)
```

6.2 Codebase nature

Palacios itself (the code in palacios/palacios) essentially consists of the following components:

- Core VMM. This code, located in palacios/palacios/{src,include}/palacios, implements the vast majority of the logic of the Palacios VMM.
- Virtual devices. This code, located in palacios/palacios/{src,include}/devices implements software versions of devices that the guest expects to be found on an x86 PC.[7]
- Host OS interfaces. This code, located in palacios/palacios/{geekos, kitten}, is a part of the integration of Palaios and the given host OS.
- Guest boot kernel. This code, located in palacios/palacios/src/vmboot, includes the software (such as a BIOS) that shall be mapped into the guest at boot time.

The Palacios build process compiles each of these components and then combines them into a single static library, palacios/palacios/build/libv3vee.a. This library can then be linked with the host OS to add VMM functionality. The host OS additionally needs to be modified to call into this library.

[7] It is very important to understand the distinction between a virtual device and a device driver. A device driver is software that implements a high-level interface to a hardware device. A virtual device is software that emulates a hardware device, exporting a low-level hardware interface to the guest. A virtual device may use the services of a device driver, but it is not a device driver. Virtual devices are a part of Palacios. Device drivers are a part of the host OS.

The top-level build process (palacios/build) continues compilation after building libv3vee.a by compiling GeekOS (or Kitten) and linking it with lib3vee.a.

The Core VMM code has essentially three components:

- Architecture independent components. These have the prefix vmm_ or vm_.
- AMD SVM-specific components. These have the prefix svm_. The AMD VMCB is in vmcb.{c,h}.
- Intel VT-specific components. These have the prefix vmx_. At the present time, SVM support is much further advanced that VT support.[8] The VT VMCB is in vmcs.{c,h}.

The file vmm.h defines the host OS interface to Palacios. Essentially, the host OS passes Palacios a set of function pointers to functionality Palacios needs. Palacios returns a set of function pointers for functions to create and control a guest. vmm_config.{c,h} implements the configuration of a guest. The main loop for guest execution is located in svm.c and vmx.c, for AMD SVM and Intel VT, respectively.

Palacios includes implementations of linked lists, queues, ring buffers, hash tables, and other basic data structures. There is generally no need to reinvent the wheel.

7 Theory of operation

We now describe, at a high-level, how Palacios works, and how it interacts with the guest and host OS.

7.1 Perspective of guest OS

The guest sees what looks to be a physical machine. The guest is booted as normal from a BIOS, and interacts directly with what appears to it to be real hardware. In its present and released form, Palacios has no hypercall interface and makes no use of paravirtualization. Interaction with the machine and its devices are through standard hardware interfaces. The guest can execute any instruction, operate in any protection ring, and manipulate any control register. In release 1.0, only 32 bit guests are supported.

7.2 Integration with host OS

Palacios is designed to be embedded into a host OS, although that host OS needs only very minimal functionality in order for Palacios to work. In release 1.0, 32 or 64 bit host OSes are supported. The host OS boots the machine, and then launches Palacios to create a guest environment. From the host OS perspective, Palacios and the guest it runs are for the most part, a kernel thread that uses typical kernel facilities. However, there are important differences:

- Palacios is involved in interrupt processing.
- Palacios can ask the host to vector interrupts to it.
- Palacios independently manages paging for guests.

[8] This is largely a function of our wanting to focus on the core of the VMM. SVM dramatically simplifies handling the early part of the guest boot process by including support for real mode. VT does not support real mode, and so emulation or V8086 mode guest functionality must be used.

- Palacios can provide guests with direct access to physical resources, such as memory-mapped and I/O-space–mapped devices.
- For certain devices (e.g. the keyboard), Palacios expects the host OS to forward device interactions (e.g. keystrokes) to it.

7.3 Boot process

The host OS is responsible for booting the machine, establishing basic drivers and other simple kernel functionality, and creating a kernel thread to run Palacios. Palacios reads a description of the desired guest configuration, and calls back into the host OS to allocate physical resources. Having the resources, Palacios establishes the initial contents of the VM control structures available on the hardware (e.g., a VMCB on AMD SVM hardware), an initial set of intercepts, and an initial set of shadow page tables. It maps into the guest's physical address space a BIOS and VGA BIOS. It then uses the SVM or VT hardware features to launch the guest, starting the guest in real mode at CS:IP=f000:fff0. Thus the guest begins executing at what looks to it like a processor reset.

As we have mapped our BIOS into these addresses (segment f000), the BIOS begins executing as one might expect after a processor reset. The BIOS is responsible for the first stage of the guest OS boot process, loading a sector from floppy, hard drive, or CD ROM and jumping to it.

7.4 VM exits and entries

The guest executes normally until an exceptional condition occurs. The occurrence of an exceptional condition causes a "VM exit". On a VM exit, the context of the guest OS is saved, the context of the host OS kernel (where Palacios is running) is restored, and execution returns to Palacios. Palacios handles the exit and then executes a VM entry, which saves the host OS context, restores the guest context, and then resumes execution at the guest instruction where the VM exit occurred. As part of the VM exit, hardware interrupts may be delivered to the host OS. As part of the VM entry, software-generated interrupts (virtual interrupts) may be delivered to the guest. At a very high level, the Palacios kernel thread handling the guest looks like this:

```
interrupts_to_deliver_to_guest = none;
guest_context = processor_reset_context;
while (1) {
(reason_for_exit, guest_context) =
                          vm_enter(guest_context,
                                   conditions,
                                   interrupts_to_deliver_to_guest);

// we are now in an exit
enable_delivery_of_interrupts_to_host();

interrupts_to_deliver_to_guest =
   handle_exit(guest_context,
                        reason_for_exit,
                        conditions);
}
```

By far, the bulk of the Palacios code is involved in handling exits.

The notion of exceptional conditions that cause VM exits is critical to understand. Exceptional conditions are generally referred to either as exit conditions (Intel) or intercepts (AMD). The hardware defines a wide range of possible conditions. For example: writing or reading a control register, taking a page fault, taking a hardware interrupt, executing a privileged instruction, reading or writing a particular I/O port or MSR, etc. Palacios decides which of these possible conditions merits an exit from the guest. The hardware is responsible for exiting to Palacios when any of the selected conditions occur. Palaios is then responsible for handling those exits.

The above description is somewhat oversimplified, as there is actually a third context involved, the shadow context. We previously used guest and shadow context interchangeably. It is important now to explain what is meant by each of host context, guest context, and shadow context. The host context is the context of the host OS kernel thread running Palacios. As one might expect, it includes register contents, control register contents, and the like. This includes the segmentation registers and paging registers. The guest context is the context in which the guest is currently running, and includes similar information. The guest OS and applications change guest context at will, although such changes may cause exits. The shadow context, which also contains similar information, is the actual processor context that is being used when the guest is running. That is, the guest does not really run in guest context, it just thinks it does. In fact, it actually runs using the shadow context, which is managed by Palacios. The VM exits are used, in part, so that Palacios can see important changes to guest context immediately, and make corresponding changes to the shadow context. A VM exit or entry is a partially hardware-managed context switch between the shadow context and the host context.

7.5 Shadow paging

From the guest's perspective, it establishes page tables that translate from the virtual addresses to physical addresses. However, Palacios cannot allow the guest to establish a mapping to any possible physical address, as this could allow the guest to conflict with Palacios, the host OS, or other guests. At the same time, Palacios must maintain the illusion that the guest is running, by itself, on a raw machine, where mappings to any physical address are permitted.

Conceptually, we can imagine using two levels of mapping to solving this problem. Virtual addresses in the guest ("guest virtual addresses") map to "physical" addresses in the guest ("guest physical addresses") using the guest's page tables, and these guest physical addresses map to "host physical addresses" (real physical addresses) using Palacios's page tables.

One way of maintaining this two level mapping is to use a hardware feature called nested paging, which is available on more recent AMD processors. Nested paging directly implements the above conceptual model as two layers of page tables, one layer controlled by the guest, and the other by the VMM. Palacios has partial support for nested paging.

Palacios's primary implementation of this abstraction is through shadow page tables. Shadow page tables are part of the shadow context—the actual processor context used when the guest is executed. They map from guest virtual addresses to host physical addresses. The guest's page tables are not used by the hardware for virtual address translation.

Changes to the guest's page tables are propagated to Palacios, which makes corresponding, but not identical, changes to the shadow page tables. This propagation of information happens through two mechanisms: page faults (which cause VM exits) and reads/writes to paging-related control registers (which cause VM exits). For example, a page fault may cause a VM exit, the handler may discover that it is due

to the shadow page tables being out of sync from the guest page tables, repair the shadow page tables, and then re-enter the guest.

Of course, some page faults need to be handled by the guest itself, and so a page fault (on the shadow page tables) which causes an exit may result in the handler delivering a page fault (on the guest page tables) to the guest. For example, a page table entry in the guest may be marked as not present because the corresponding page has been swapped to disk. An access to that page would result in a hardware page fault, which would result in a VM exit. The handler would notice that the shadow page table entry was in sync with the guest, and therefore the guest needed to handle the fault. It would then assert that a page fault for the guest physical address that originally faulted should be injected into the guest on the next entry. This injection would then cause the guest's page fault handler to run, where it would presumably schedule a read of the page from disk.

The core idea behind Palacios's current shadow paging support is that it is designed to act as a "virtual TLB". A detailed description of the virtual TLB approach to paging in a VMM is given in the Intel processor documentation. One issue with virtual TLBs is that switching from one page table to another can be very expensive, as the "virtual TLB" is flushed. Palacios attempts to ameliorate this problem through the use of a shadow page table cache that only flushes tables as they become invalid, as opposed to when they are unused. This is a surprisingly subtle problem since the page tables themselves, are, of course, mapped via page tables.

7.6 Interrupt delivery

Interrupt delivery in the presence of a VMM like Palacios introduces some complexity. By default Palacios sets the "interrupt exiting" condition when entering a guest. This means that any hardware interrupt (IRQ) that occurs while the guest is running causes an exit back to Palacios. As a side-effect of the exit, interrupts are masked. Palacios turns off interrupt masking at this point, which results in the interrupt being delivered by the common path in the host OS. Palacios has the option to translate the interrupt before unmasking, or even can ignore it, but it does not currently do either.

A Palacios guest can be configured so that it receives particular hardware interrupts. For this reason, the host OS must provide a mechanism through which Palacios can have the interrupt delivered to itself, as well as any other destination. When such a "hooked" interrupt is delivered to Palacios, it can arrange to inject it into the guest at the next VM entry. Note that this injected interrupt will not cause another exit.

Similar to a guest, a virtual device linked with Palacios itself can arrange for interrupts to be delivered to itself via a callback function. For example, a virtual keyboard controller device could hook keyboard interrupts for the physical keyboard so that it receives keystroke data.

The Palacios convention, however, is generally to avoid direct interrupt hooking for virtual devices. Instead, we typically have virtual devices export custom callback functions that can then be called in the relevant place in the host, at the host's convenience. For example, our keyboard controller virtual device works like the following. When the physical keyboard controller notes a key-up or key-down event, it generates an interrupt. This interrupt causes an exit to Palacios. Palacios turns on interrupts, which causes a dispatch of keystroke interrupt to the host OS's keyboard device driver. The driver reads the keyboard controller, and then pushes the keystroke data to the host OS. The host OS is responsible for demultiplexing between its own processes/threads, and the Palacios threads. If it determines that the keystroke belongs to Palacios, it calls a "deliver keystroke" upcall in Palacios, which, after further demultiplexing, vectors to the relevant virtual keyboard controller device. This virtual device processes the data, updates its internal state, and (probably) requests that Palacios inject an interrupt into the guest on the next entry. The guest driver

will, in turn, respond to the interrupt by attempting to read the relevant I/O port, which will again cause an exit and vector back to the virtual keyboard controller.

7.7 Device drivers and virtual devices

Device drivers exist at two levels in Palacios. First, the host OS may have device drivers for physical hardware. Second, the guest OS may have device drivers for physical hardware on the machine, and for virtual devices implemented in Palacios. A virtual device is a piece of software linked with Palacios that emulates a physical device, perhaps making use of device drivers and other facilities provided by the host OS. From the guest's perspective, physical and virtual devices appear identical.

The implementation of virtual devices in Palacios is facilitated by its shadow paging support, I/O port hooking support, and interrupt injection support. Palacios's shadow paging support provides the ability to associate ("hook") arbitrary regions of guest physical memory addresses with software handlers that are linked with Palacios. Palacios handles the details of dealing with the arbitrary instructions that can generate memory addresses on x86. This is the mechanism used to enable memory-mapped virtual devices. Similarly, Palacios allows software handlers to be hooked to I/O ports in the guest, and it handles the details of the different kinds of I/O port instructions (size, string, rep prefix, etc) the guest could use. This is the mechanism used to enabled I/O-space mapped virtual devices. As we previously discussed, Palacios handlers can intercept hardware interrupts, and can inject interrupts into the guest. This provides the mechanism needed for interrupt-driven devices.

In addition to hooking memory locations or I/O ports, it is also possible in Palacios to allow a guest to have direct access, without exits, to given physical memory locations or I/O ports. This, combined with the ability to revector hardware interrupts back into the guest, makes it possible to assign particular physical I/O devices directly to particular guests. While this can achieve maximum I/O performance (because minimal VM exits occur) and maximum flexibility (because no host device driver or Palacios virtual device is needed), it requires that (a) the guest be mapped so that its guest physical addresses align with the host physical addresses the I/O device uses, and (b) that we trust the guest not to ask the device to read or write memory the guest does not own.

8 Host OS interface

Palacios expects to be able to request particular services from the OS in which it is embedded. Function pointers to these services are supplied in a `v3_os_hooks` structure:

```
struct v3_os_hooks {
  void (*print_info)(const char * format, ...);
  void (*print_debug)(const char * format, ...);
  void (*print_trace)(const char * format, ...);

  void *(*allocate_pages)(int numPages);
  void (*free_page)(void * page);

  void *(*malloc)(unsigned int size);
  void (*free)(void * addr);
```

```
  void *(*paddr_to_vaddr)(void *addr);
  void *(*vaddr_to_paddr)(void *addr);

  int (*hook_interrupt)(struct guest_info * vm, unsigned int irq);

  int (*ack_irq)(int irq);

  unsigned int (*get_cpu_khz)(void);

  void (*start_kernel_thread)(void); // include pointer to function

  void (*yield_cpu)(void);

};
```

The `print` functions are expected to take standard `printf` argument lists. Generally speaking, `print_info()` is called for output that is unrelated to debugging or performance evaluation, while `print_debug()` and `print_trace()` are called for these latter reasons.

`allocate_pages()` is expected to allocate contiguous physical memory, specifically `numPages` 4 KB pages, and return the physical address of the memory. `free_page()` deallocates a physical page at a time. This interface is in flux as we incorporate better support for guest superpages and PAE, as well as 64 bit guest support. `malloc()` and `free()` should allocate kernel memory and return virtual addresses suitable for use in kernel mode.

The `paddr_to_vaddr()` and `vaddr_to_paddr()` functions should translate from host physical addresses to host virtual addresses and from host virtual addresses to host physical addresses, respectively.

The `hook_interrupt()` function is how Palacios requests that a particularly interrupt should be vectored to itself. When the interrupt occurs, the host OS should use `v3_deliver_irq()` to actually push the interrupt to Palacios. Palacios will acknowledge the interrupt by calling back via `ack_irq()`.

`get_cpu_khz()` and `start_kernel_thread()` are self-explanatory. The Palacios guest execution thread will call `yield_cpu()` when the guest does not currently require the CPU. The host OS can, of course, also preempt it, as needed.

After filling out a `v3_os_hooks` structure, the host OS calls `Init_V3()` with pointers to this structure and to a `v3_ctrl_ops` structure. In response, Palacios will determine the hardware virtualization features of the machine, and, if possible, will populate the `v3_ctrl_ops` structure. This structure is as follows:

```
struct v3_ctrl_ops {
  struct guest_info *(*allocate_guest)(void);

  int (*config_guest)(struct guest_info * info,
                      struct v3_vm_config *config_ptr );
  int (*init_guest)(struct guest_info * info);
  int (*start_guest)(struct guest_info * info);

  int (*has_nested_paging)(void);
```

```
};
```

The host OS then allocates, configures, initializes, and starts a guest VM using the corresponding calls. `start_guest()` returns when the guest terminates. The `guest_info` structure should be treated as opaque by the host OS.

The `v3_vm_config` structure is currently in a state of flux. It contains basic information such as where the initial `vm_kernel` components are, and whether ramdisk boot facilities are to be used. At this point, we manage most of the guest configuration in a relatively hard-coded manner (see `vmm_config.c:v3_config_guest()`). This will be changed soon to a general configuration process where the host will simply pass `config_guest()` a pointer to a text representation of the needed guest configuration.

As can be seen, the bare outline of the interface between the host OS and Palacios is quite simple. However, there *is* complexity in the Palacios/host OS interaction vis a vis interrupts (covered in Section 7), and in pushing necessary data to Palacios for use in virtual devices (covered in Section 9).

9 Virtual devices

Every modern OS requires a given set of hardware devices to function, for OSes running in a virtual contexts the set of required devices must be virtualized. The Palacios VMM contains a framework to make implementation of these virtual devices easier. Palacios also includes a set of essential virtual devices. These are located in the device directory of the Palacios code tree, which is also where additional virtual devices should be placed.

9.1 Included virtual devices

The current list of included virtual devices is:

- NVRAM and RTC
- Keyboard/Mouse [PS2]
- Programmable Interrupt Controller (PIC) [based on Intel 8259]
- Programmable Interval Timer (PIT) [based on Intel 8024]
- IDE bus
- RAM-based emulated ATAPI CD ROM
- Debug port for the BOCHS BIOS
- A generic interception device to inspect I/O traffic and IRQ occurrences.

9.2 Virtual device interface

Virtual devices are required to export a common interface to Palacios that allows the VMM to control the devices. This interface is defined in `struct vm_device`. This structure is created by the VMM with arguments provided by the device's creation function.

Currently each device is required to export a create function via its header file. This function is called by the VMM whenever it wants to create the given device for a specific VM. The function should return a `struct vm_device *` that will be used to reference the device instance from then on. The `struct vm_device` has several fields that are important to note:

- `char name[32]` Contains the name of the device
- `void * private_data` This opaque pointer that can point to any value. Typically, it points to the internal state of the device instance.
- `struct guest_info * vm` A pointer to the VM that the device instance is associated with.

The functional interface is exported via a structure of function pointers:

```
struct vm_device_ops {
  int (*init)(struct vm_device *dev);
  int (*deinit)(struct vm_device *dev);
  int (*reset)(struct vm_device *dev);
  int (*start)(struct vm_device *dev);
  int (*stop)(struct vm_device *dev);
};
```

This structure is usually declared statically for each virtual device implementation. For instance a device called "my_device" would declare:

```
static struct vm_device_ops my_device_dev_ops = {
  .init = my_device_init_fn,
  .deinit = my_device_deinit_fn,
  .reset = my_device_reset_fn,
  .start = my_device_start_fn,
  .stop = my_device_stop_fn,
};
```

To create an instance of a virtual device, all of these structures are used:

```
struct vm_device * v3_create_device(char * name,
                                    struct vm_device_ops * ops,
                                    void * private_data);
```

When `v3_create_device()` is called, the VM with which the device will be associated is not yet valid and so must not be referenced. It is for this reason that each device must also implement an `init` and `deinit` function through the `vm_device_ops` structure. When these are called, the VM is valid.

Once a VM has been fully instantiated and is ready to be started, the VMM will call each device's `init()` function. At this point the device can configure itself to correctly interact with the guest's environment. The methods for guest interaction are discussed in detail in Section 10. Typically, however, devices hook themselves to IO ports, guest physical memory regions, and sometimes interrupts. These hooks are used to give the device a presence in the "hardware" the guest sees. For example, when the guest code reads from an I/O port that has been hooked by a particular virtual device, Palacios calls back to that function that that device provided to complete the read.

Virtual devices typically also need to interact with the host OS, typically to access physical devices on which they are built. This interaction is usually quite specific to the kind of virtual device and to the host OS. For the virtual devices we provide along with Palacios, we have defined such interfaces. Therefore, these devices provide good examples for how such interaction should be coded.

10 Guest interaction

There are numerous ways for Palacios components, such as virtual devices or experimental systems software, to interact with a VM guest environment and control its behavior. Specifically, Palacios allows components to dynamically intercept IRQs, I/O port operations, MSR operations, and memory operations. They can also intercept certain specific host events. Components can also inject interrupts into running guests. We will now describe these interfaces and their APIs.

10.1 Interrupt hooking

Interrupt hooking is useful when a Palacios component wants to handle specific interrupts that occur in guest context. For example, if a special hardware device is being used exclusively by Palacios (and not the host OS) then Palacios can request that the device's interrupts be delivered directly to the responsible Palacios component. This is done by calling `v3_hook_irq()`.

```
int v3_hook_irq(struct guest_info * info,
                uint_t irq,
                int (*handler)(struct guest_info * info,
                               struct v3_interrupt * intr,
                               void * priv_data),
                void  * priv_data);
```

This function takes a reference to a guest context, an IRQ number, and a pointer to a function that will be responsible for handling the interrupt. The function also allows arbitrary state to be made available to the handler via the `priv_data` argument. `v3_hook_irq()` installs an interrupt handler in the host os that will then call the specified handler whenever it is triggered. The arguments passed to the handler are a pointer to the guest context, a pointer to the interrupt state, and the pointer to the arbitrary data originally passed into the hook function.

The interrupt state is included in the `v3_interrupt` structure:

```
struct v3_interrupt {
  unsigned int irq;
  unsigned int error;
  unsigned int should_ack;
};
```

It includes the interrupt number, the error number that is optionally included with some interrupts, as well as a flag specifying whether the host OS should acknowledge the interrupt when the handler returns.

Another use case for interrupt hooking is the situation where Palacios wants to give a guest complete control over a piece of hardware. This requires that any interrupts generated by the actual hardware device be propagated into the guest environment. While a Palacios component could register a handler for the interrupt and manually resend it to the guest, there is a shortcut for this situation:

```
int v3_hook_passthrough_irq(struct guest_info * info, uint_t irq);
```

This function tells Palacios to register an interrupt handler with the host, and then inject any interrupt that occurs directly into the guest.

Interrupts can be dynamically hooked and unhooked.

10.2 I/O hooking

I/O hooking is essential for building many forms of virtual devices. The x86 architecture includes a 16-bit "I/O" address space (the addresses are called "I/O ports") that is accessed with a special set of IN and OUT instructions. I/O hooking allows a Palacios component, such as a virtual device, to register a handler for particular ports. Whenever the guest OS performs an IN or OUT operation on a I/O port, that operation is translated to a read or write call to a specified handler.

The API looks like this:

```
void v3_hook_io_port(struct guest_info * info, uint_t port,
                     int (*read)(ushort_t port, void * dst,
                                 uint_t length, void * private_data),
                     int (*write)(ushort_t port, void * src,
                                  uint_t length, void * private_data),
                     void * private_data);
```

For each port to be intercepted a call is made to `v3_hook_io_port` with the port number and two function pointers to handle the reads/writes. Each I/O operation will result in one call to the appropriate read/ write function. The read call corresponds to an IN instruction and reads data into the guest, while the write call corresponds to OUT instructions and writes data from the guest.

IN and OUT instructions come in byte, word (double byte), and double word (quad byte) varieties. These are translated into equivalent length calls to the read and write handlers. The handlers are expected to handle them in one step. There are also string versions of the IN and OUT instructions. And, furthermore, the REP prefix can be used. Repeated forms of the string derivatives result in multiple calls to the read or write function for each repetition. Note that this is how it is done in hardware as well. The calls to the read and write handlers correspond to what a device would see on the bus.

The `v3_unhook_io_port()` function removes an I/O port hook. I/O ports can be dynamically hooked and unhooked.

10.3 Memory hooking

Memory hooking is essential for building many forms of virtual devices, in particular those that are memory-mapped. Because memory hooking can be done in Palacios at the byte granularity, it is useful well beyond virtual devices.

Palacios supports intercepting read and write operations to specific memory regions defined at byte address granularity. This is done via a call to `v3_hook_guest_mem`:

```
int v3_hook_guest_mem(struct guest_info * info,
                      addr_t guest_addr_start,
                      addr_t guest_addr_end,
                      int (*read)(addr_t guest_addr, void * dst,
```

```
                                uint_t length, void * priv_data),
                    int (*write)(addr_t guest_addr, void * src,
                                 uint_t length, void * priv_data),
                    void * priv_data);
```

The `guest_addr_start` and `guest_addr_end` arguments specify the guest physical memory region for which read and write operations will be intercepted and sent to the handler functions. The `guest_addr` parameters to the read/write handlers are the guest physical addresses at which the operations were made.

The read and write handlers are called by Palacios in response to individual memory references in the guest. For typical uses of memory hooking, this usually amounts to instruction level granularity, as almost all instructions only support a single memory operand, and it's rare that one hooks memory containing instructions. For example, a `mov eax, dword <addr>` instruction would result in a 4 byte read call. It is important to note that almost any instruction on the x86 can include a memory operand. Palacios handles this complexity automatically.

The current implementation of memory hooking is very specific to SVM and 32 bit page tables. Also be aware that each memory operation currently takes two VM exits and entries. Palacios currently uses a technique we developed that avoids having to emulate or even fully decode the faulting instruction.

Memory regions can be dynamically hooked and unhooked.

10.4 MSR hooking

Model-specific registers (MSRs) are registers that the guest can access to probe, measure, and control the specific microprocessor it is running on. Palacios components can hook reads and writes to MSRs. This functionality can be used to hide microprocessor functionality from the guest or provide a software implementation of that functionality.

The MSR hooking framework looks like this:

```
int v3_hook_msr(struct guest_info * info, uint_t msr,
                int (*read)(uint_t msr,
                            struct v3_msr * dst,
                            void * priv_data),
                int (*write)(uint_t msr,
                             struct v3_msr src,
                             void * priv_data),
                void * priv_data);
```

Similar to other hooks, the handlers are called when the hooked MSR is read or written. Palacios handles the details of the particular instruction that is accessing the MSR.

The handler is passed the MSR as a union:

```
struct v3_msr {
  union {
    ullong_t value;
    struct {
      uint_t lo;
      uint_t hi;
    }       splitvalue;
```

```
  };
};
```

10.5 Host event hooking

To allow specific host OS events to be forwarded to appropriate Palacios components, Palacios implements a mechanism for those components to register to receive given host events. Currently, Palacios supports keyboard, mouse, and timer event notifications. These events must be generated from inside the host OS, and currently target very specific interfaces that cannot be generalized. The intent is to allow for greater flexibility in designing the host interface to the Palacios VMM.

The method of registering for a host event is through the function `v3_hook_host_event`:

```
int v3_hook_host_event(struct guest_info * info,
                       v3_host_evt_type_t event_type,
                       union v3_host_event_handler handler,
                       void * private_data);
```

The `event_type` and `handler` arguments are defined as:

```
typedef enum {HOST_KEYBOARD_EVT,
              HOST_MOUSE_EVT,
              HOST_TIMER_EVT} v3_host_evt_type_t;

union v3_host_event_handler {
  int (*keyboard_handler)(struct guest_info * info,
                          struct v3_keyboard_event * evt,
                          void * priv_data);
  int (*mouse_handler)(struct guest_info * info,
                       struct v3_mouse_event * evt,
                       void * priv_data);
  int (*timer_handler)(struct guest_info * info,
                       struct v3_timer_event * evt,
                       void * priv_data);
};
```

As one can imagine, the current set of event types and handlers supports the standard virtual devices included with Palacios. A shortcut exists to cast a given handler function to a `union v3_host_event_handler` type:

```
#define V3_HOST_EVENT_HANDLER(cb) ((union v3_host_event_handler)cb)
```

To hook a host event, a Palacios component firsts implements a handler as defined in the `v3_host_event_handler` union. Then it calls `v3_hook_host_event` with the correct `event_type` as well as the handler function, cast appropriately. As an example, a component that wants to hook mouse events would look like:

```
int mouse_event_handler(struct guest_info * info,
                        struct v3_mouse_event * evt,
```

```
                          void * private_data) {
   // implementation
}

v3_hook_host_event(dev->vm, HOST_MOUSE_EVT,
                   V3_HOST_EVENT_HANDLER(mouse_event_handler),
                   dev);
```

Each type of event handler takes a different argument. These are the current arguments for the currently supported events:

```
struct v3_keyboard_event {
  unsigned char status;
  unsigned char scan_code;
};

struct v3_mouse_event {
  unsigned char data[3];
};

struct v3_timer_event {
  unsigned int period_us;
};
```

The keyboard event includes the current status of the keyboard controller and the raw scan code (if any). The mouse event includes the latest mouse data packet. The timer event includes the time since the previous timer event.

Host events can be dynamically hooked and unhooked.

The host event hooking infrastructure is likely to change over time.

10.6 Injecting interrupts and exceptions

A Palacios component can decide to inject an exception or an interrupt into the guest. As explained earlier, in the theory of operation of Palacios, injection occurs as a part of entry into the guest. Injected interrupts or exceptions do *not* cause an exit, although the guest's processing of them may do so.

There are two functions that can be used to raise an exception:

```
int v3_raise_exception(struct guest_info * info,
                       uint_t excp);
int v3_raise_exception_with_error(struct guest_info * info,
                                  uint_t excp,
                                  uint_t error_code);
```

These functions are self-explanatory. Which one should be used depends on the specific exception that is being raised (only some exceptions include an error code). It is the calling Palacios component's responsibility to use the appropriate function.

Interrupts can be raised and lowered:

```
int v3_raise_irq(struct guest_info * info, int irq);
int v3_lower_irq(struct guest_info * info, int irq);
```

It is important to note that raising an interrupt raises it on the virtual PIC. When the interrupt is actually delivered depends on the guest's current configuration of the PIC, whether it has interrupts masked or turned off, and the priorities of other interrupts that are also raised. Priorities between interrupts and interrupts and exceptions are strictly observed. Exceptions and interrupts work the same as on the actual PC hardware.

11 Coding guidelines

Use whitespace well. "The use of equal negative space, as a balance to positive space, in a composition is considered by many as good design. This basic and often overlooked principle of design gives the eye a 'place to rest,' increasing the appeal of a composition through subtle means." Use spaces not tabs.

Stop on failure. Because booting even a basic Linux kernel results in over 1 million VM exits, catching silent errors is next to impossible. For this reason *ANY* time your code has an error it should return -1, and expect the execution to halt shortly afterwards. This includes unimplemented features and unhandled cases. These cases should *ALWAYS* return -1.

Keep the namespace small and pure. To ease porting, externally visible function names should be used rarely and have unique names. Currently we have several techniques for achieving this:

1. `#ifdefs` in the header file. As Palacios is compiled, the symbol `__V3VEE__` is defined. Any function that is not needed outside the Palacios context should be inside an `#ifdef __V3VEE__` block. This will make it invisible to the host OS.

2. `static` functions. Any utility functions that are only needed in the .c file where they are defined should be declared as `static` and not included in the header file.

3. `v3_` prefix. Major interface functions should be named with the prefix `v3_`. This makes it easier to understand how to interact with the subsystems of Palacios. Further, if such functions need to be externally visible to the host OS, the prefix makes the unlikely to collide with host OS symbols.

Respect the debugging output convention. Debugging output is sent through the host os via functions in the `os_hooks` structure. These functions should be called via their wrapper functions, which have simple `printf` semantics. Two functions of note are `PrintDebug` and `PrintError`.

- `PrintDebug` should be used for debugging output that can be turned off selectively by the compile-time configuration.

- `PrintError` should be used when an error occurs. These calls will never be disabled and will always print.

12 How to contribute

If you have modified or extended Palacios and would like to contribute your changes back to the project, you can do so using git's patch facility.

When code is ready to be contributed, you should first do a local commit:

```
git commit
```

Next, generate a patch set that can be applied to the remote repository. This is done by running

```
git format-patch origin/<branch-name>
```

where <branch-name> will be release-1.0 unless otherwise noted by us, or by the v3vee web site.

This will find all of the commits you have done locally and that are not present in the remote branch. It will then write out a series of patch files in the current directory containing all of the local commits. You can then email these patch files to contributions@v3vee.orgfor possible inclusion in the mainline codebase.

13 Acknowledgments

Lei Xia integrated Palacios and XED, and developed the LWIP and UIP-based network stacks. Matt Wojcik and Peter Kamm implemented the NE2K and RTL8139 drivers. Zheng Cui of the University of New Mexico developed the RAM Disk support in Palacios. Trammell Hudson and Kevin Pedretti of Sandia National Labs helped us to rationalize some of the Palacios build process. With their help, Palacios can now run on top of Sandia's Kitten operating system. The code is included in release 1.0 and will integrate with the upcoming Kitten public release.

The efforts of the V3VEE advisory board have been very useful.

This project is made possible by support from the National Science Foundation (NSF) via grants CNS-0709168, CNS-0707365, and the Department of Energy (DOE) via a subcontract from Oak Ridge National Laboratory (ORNL) on grant DE-AC05-00OR22725. Jack Lange was partially supported by a Symantec Research Labs Fellowship.

References

[1] AMD CORPORATION. AMD64 virtualization codenamed "pacific" technology: Secure virtual machine architecture reference manual, May 2005.

[2] BELLARD, F. Qemu: A fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track* (April 2005).

[3] DUNKELS, A. Full tcp/ip for 8-bit architectures. In *Proceedings of the first international conference on mobile applications, systems and services (MOBISYS)* (May 2003).

[4] HOVENMEYER, D., HOLLINGSWORTH, J., AND BHATTACHARJEE, B. Running on the bare metal with geekos. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE)* (2004).

[5] INTEL CORPORATION. Intel virtualization technology specification for the ia-32 intel architecture, April 2005.

[6] LAWTON, K. Bochs: The open source ia-32 emulation project. http://bochs.sourceforge.net.

[7] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementatin (PLDI)* (June 2005).

[8] RESEARCH HYPERVISOR TEAM. The research hypervisor. http://www.research.ibm.com/hypervisor/, 2005.

[9] SANDIA NATIONAL LABORATORIES. Kitten operating system. Soon to be available.

[10] UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A., MARTIN, F., ANDERSON, A., BENNETTT, S., KAGI, A., LEUNG, F., AND SMITH, L. Intel virtualization technology. *IEEE Computer* (May 2005), 48–56.