An Introduction to the Palacios Virtual Machine Monitor---Version 1.3

Jack Lange Peter Dinda Kyle Hale Lei Xia

Abstract

Palacios is a virtual machine monitor (VMM) from the V3VEE Project that is available for public use as a community resource. Palacios is highly configurable and designed to be embeddable into different host operating systems, such as Linux and the Kitten lightweight kernel. Palacios is a non-paravirtualized VMM that makes extensive use of the virtualization extensions in modern Intel and AMD x86 processors. Palacios is a compact codebase, consisting of ~96, 000 lines of C and assembly of which ∼40, 000 are in the core VMM, and the remainder are in virtual devices, extensions, an overlay network, and other optional features. Palacios is designed to be easy to understand and readily configurable for different environments. It is unique in being designed to be embeddable into other OSes instead of being implemented in the context of a specific OS. This document describes the overall structure of Palacios and how it works, with a focus on the Linux kernel module variation (other embeddings, such as for Kitten and Minix are documented separately). We also explain how to download and run Palacios, develop within it, and extend it.

This project is made possible by support from the National Science Foundation (NSF) via grants CNS-0709168 (Northwestern University), CNS-0707365 (University of New Mexico), and the Department of Energy (DOE) via DE-SC0005343 (Northwestern University, University of Pittsburgh, University of New Mexico, Sandia National Labs, and Oak Ridge National Labs). Initial support for the development of Palacios was provided to Northwestern University via a subcontract from Oak Ridge National Laboratory on grant DE-AC05-00OR22725. Sandia National Laboratory provided seed funding to Northwestern University for multicore guest support. 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 1.3

Jack Lange Peter Dinda Kyle Hale Lei Xia

Abstract

Palacios is a virtual machine monitor (VMM) from the V3VEE Project that is available for public use as a community resource. Palacios is highly configurable and designed to be embeddable into different host operating systems, such as Linux and the Kitten lightweight kernel. Palacios is a non-paravirtualized VMM that makes extensive use of the virtualization extensions in modern Intel and AMD x86 processors. Palacios is a compact codebase, consisting of $\sim 96,000$ *lines of C and assembly of which* $\sim 40,000$ *are in the core VMM, and the remainder are in virtual devices, extensions, an overlay network, and other optional features. Palacios is designed to be easy to understand and readily configurable for different environments. It is unique in being designed to be embeddable into other OSes instead of being implemented in the context of a specific OS. This document describes the overall structure of Palacios and how it works, with a focus on the Linux kernel module variation (other embeddings, such as for Kitten and Minix are documented separately). We also explain how to download and run Palacios, develop within it, and extend it.*

The author list reflects the writers of this technical report. The Palacios codebase has numerous contributors, as described briefly in Section 16 and on the V3VEE web site (`v3vee.org`).

This project is made possible by support from the National Science Foundation (NSF) via grants CNS-0709168 (Northwestern University), CNS-0707365 (University of New Mexico), and the Department of Energy (DOE) via DE-SC0005343 (Northwestern University, University of Pittsburgh, University of New Mexico, Sandia National Labs, and Oak Ridge National Labs). Initial support for the development of Palacios was provided to Northwestern University via a subcontract from Oak Ridge National Laboratory on grant DE-AC05-00OR22725. Sandia National Laboratory provided seed funding to Northwestern University for multicore guest support. Jack Lange was partially supported by a Symantec Research Labs Fellowship.

Contents

1 Introduction

The V3VEE project (`v3vee.org`) has created a virtual machine monitor framework for modern architectures (those with hardware virtualization support) that enables the compile-time creation of VMMs with different structures, including those optimized for computer architecture research and use in high performance computing. V3VEE began as a collaborative project between Northwestern University and the University of New Mexico. It currently includes Northwestern University, the University of New Mexico, the University of Pittsburgh, Sandia National Labs, and Oak Ridge National Lab.

Palacios[1] is the V3VEE project's publicly available, open-source VMM. Palacios currently targets x86 and x86_64[2] architectures (hosts and guests) and relies on the use of AMD SVM [1] or Intel VT[3, 9] architectural extensions. It provides both shadow paging in the virtual MMU model, as well as nested paging on Intel and AMD hardware that implement this feature. Palacios runs directly on the hardware and provides a non-paravirtualized interface to the guest. An extensive infrastructure for hooking guest physical memory, guest I/O ports, and interrupts facilitates experimentation.

Palacios is designed as an embeddable VMM that can be integrated into different host operating systems. This document describes the Linux embedding of Palacios, in which Palacios is compiled into a kernel module that can be inserted into a running Linux kernel. Palacios is also commonly embedded (via static linking) into Sandia's Kitten lightweight OS, for use in research in virtualization for high performance computing. Embeddings for the Minix 3 operating system and the GeekOS teaching OS have also been developed. The primary guest operating systems used for testing Palacios are Linux, CNL, Catamount, and others.

Research publications on or using Palacios can be found on the V3VEE project web site. A discussion of our and others' Palacios-based research is beyond the scope of this document, but the following research papers also contain further relevant information about the structure of Palacios:

- For more information on the structure of the core VMM and for information on Palacios's integration with the Kitten lightweight kernel for virtualization of high performance computers: J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, R. Brightwell, *Palacios and Kitten: New High Performance Operating Systems for Scalable Virtualized and Native Supercomputing*, Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010), April, 2010.

- For more information on Palacios's adaptations specifically for virtualization at large scales, as well as a scaling study to over 4096 nodes: J. Lange, K. Pedretti, P. Dinda, P. Bridges, C. Bae, P. Soltero, A. Merritt, *Minimal Overhead Virtualization of a Large Scale Supercomputer*, Proceedings of the 2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2011), March, 2011.

- For more information on the VNET/P high performance overlay network component: L. Xia, Z. Cui, J. Lange, Y. Tang, P. Dinda, P. Bridges, *VNET/P: Bridging the Cloud and High Performance Computing Through Fast Overlay Networking*, Technical Report NWU-EECS-11-07, Department of Electrical Engineering and Computer Science, Northwestern University, July, 2011.

[1]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.

[2]32 and 64 bit guest operating systems are supported, as are 32 and 64 bit host operating systems. 32 bit host Linux is currently deprecated.

You may also find the following additional documents to be helpful. They are available from the Palacios portion of the V3VEE website.

- A getting started guide to Palacios.
- A document describing how to set up PXE network booting to test operating systems. This is an essential to being able to achieve a fast edit/compile/test cycle when testing on physical hardware. The document is applicable to Palacios in Kitten and to Palacios in Linux. Y. Tang, L. Xia, *Booting Palacios/Kitten or Palacios/Linux Over the Network Using PXE.*
- A guide to integrating Palacios 1.3 with the Kitten 1.3 release.

The 1.3 version of Palacios is a compact codebase. The core VMM, including both AMD and Intel support, contains about 40,000 lines of C and assembler. Beyond this, virtual devices, extensions, the VNET overlay network, and other optional features add about another 56,000 lines of C and assembler. The size of the host OS interface code depends on the set of features that are implemented. The host OS interface contained in the Linux kernel module is the largest such code we have written. It comprises about 10,000 lines of C, with an additional 4,000 lines of C for the userspace tools.

The purpose of this document is to explain the overall structure and operation of Palacios, and to act as a guide for those who would like to use, develop within, 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

In order to run Palacios you will need either suitable hardware or an emulator, as well as a small set of required build software. Almost any recent machine can be used for testing, and if none is available a free emulator can be used. The needed build software is all free and is already a part of typical Linux distributions, or can be readily installed.

Hardware or emulator for testing To run the current version of Palacios, you will need either a physical machine that supports the AMD SVM extensions[3], the Intel VT extensions[4], or a software emulator that supports one of these. Appendix A describes the common test hardware that we use in the project.

For emulation, we recommend using the excellent QEMU emulator [2], which is freely available. QEMU can emulate the x86 ISA on a variety of processors and does so particularly well when run on another x86 chip. We currently use QEMU versions 0.10.5, 0.14.1, and 0.15. These versions of QEMU include an implementation of the AMD SVM hardware extensions (nested paging is not supported). Because QEMU is an emulator, you do not actually need a physical machine that supports *any* hardware virtualization extensions. Indeed, you do not even need a machine with an x86 processor to to test Palacios. QEMU runs on Windows, Linux, Mac OS X, and most other Unix platforms.

When using physical hardware, we find it convenient to either work directly on a local machine or use a Preboot Execution Environment (PXE) with remote development platforms. PXE provides BIOS-level support for booting machines over the network. One can easily set up a server machine that provides the

[3]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.

[4]Similarly, Palacios will work with, or without the Intel EPT hardware nested paging feature.

Palacios image via PXE. Combined with a network power switch or RSA card and an IP-KVM, PXE boot support makes it straightforward to develop and test on remote hardware.

There is a trade-off between using physical hardware and emulation to run and test Palacios. Physical hardware is much faster than emulation, but emulation can provide a more deterministic environment with better tools for debugging. Also, it is common in Palacios, and other OSes, to employ "printf debugging" with output going to a serial port, or a kernel debugger that attaches to the kernel via a serial port. QEMU's emulated serial port mostly ignores speed settings and operates as fast as possible. In comparison, a hardware serial port is typically limited to 115 kbps. For this reason, the highly "printf-intensive" code typically used when debugging often runs *faster* on QEMU than on physical hardware.

Essential software for building Palacios The software environment needed to build Palacios includes several open source or free software tools that you will need to have installed in order to get started. Typical Linux distributions already have these tools installed. The basic set of 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.
- **GNU make**.
- **git**. As we use this for version control, you will need to have it installed if you intend to use the public repository or contribute code to the development branch.
- *(When building a Palacios Linux kernel module only)* Linux kernel sources or at the kernel header files for the Linux kernel version you will be using. Most Linux distributions make the headers available as part of a *kernel-headers* package.

As an example, at the time of this writing, a frequently used compile server in our group has a vanilla Red Hat Enterprise Linux Server 5.4, which provides "gcc-4.1.2-46.el5_4.2" , "binutils-2.17.50.0.6-12.el5", "make-3.81-3.el5" and "git-1.5.5.6-4.el5". A frequently used host OS kernel for testing is Linux 2.6.32, downloaded from kernel.org. We also make extensive use of Fedora 15, which functions with Palacios out of the box with no additional configuration necessary.

Non-essential software If you plan to do testing on a physical machine, you may find *kermit* to be helpful, as well as utilities like *setserial* and *stty* in order to control your serial port. Most host OSes, including the ones we support, such as Linux, send debugging output to the first serial port ("COM1"). We have also found a properly configured terminal server to be extremely helpful, as it allows developers to just telnet to the relevant serial connection.

Tools for walking the codebase can be helpful. Some developers advocate *cscope* and *ctags*. Others use *etags* and *emacs*. In general, there is a wide array of plug-ins available for both *emacs* and *vim* to aid in source code navigation. It is also possible to develop Palacios from within Eclipse, although we do not provide the project files to do this.

Especially when used as a Linux kernel module, gdb can be a very helpful tool when remotely debugging Palacios. This is not as powerful as using *gdb* on an application, but useful nonetheless. Even if the host OS you embed Palacios into does not support *kgdb*, it is still possible to use gdb for remote debugging on an emulated machine using QEMU's features. We discuss debugging Palacios with these tools in detail in Sections 6.6 and 6.7

Linux host kernel requirements A 2.6 kernel is required for use as the host OS with the Palacios Linux kernel module. The earliest version we have used is 2.6.30. We commonly test with 2.6.32 and 2.6.40 aka 3.0. Although Palacios supports compilation for a 32 bit host OS, 32 bit Linux host support is currently deprecated. A 64 bit Linux host is highly recommended. Fedora 15 runs Palacios out of the box using the stock configuration. We have also verified that Palacios will load into a Red Hat Enterprise Linux 6.0 system running a 2.6.32 kernel built from the RHEL sources using the kernel RPM build process with the distribution's config file.

As we elaborate on later, **the Linux host kernel MUST be configured with the memory hot-remove feature.** This feature is located under Processor type and features → Allow for memory hot-add → Allow for memory hot-remove. **It is NOT enabled by default in most kernels.** If this feature is not enabled, Palacios will be unable to allocate large chunks of memory and VM creation will fail.

3 How to get the code

Palacios is available from the V3VEE Project's web site at `http://www.v3vee.org/palacios`. 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 gain access to the latest public version of Palacios and to contribute to its development. You can access the repository using both standard git tools and a web browser. The public git repository tracks our internal development repository with a delay of 30 minutes. It provides access to the most up to date Palacios code that is publicly available.

- **Source releases and snapshots:** You can download source releases and snapshots from our web site.

The Palacios tar file or git checkout includes some of the following:

- **BOCHS** [5]: Modified versions of the BOCHS ROM BIOS and VGA BIOS are included for bootstrapping the guest VM. This is the default initial guest boot environment.

- **VM86** [7]: IBM's virtual 8086 mode emulator is included to provide real-mode guest bootstrap support on Intel VT machines. It is unused on AMD machines.

- **XED** [6]: Intel's XED x86 instruction decoder/encoder from the Pin Project is included to optionally provide complete x86 instruction decoding capabilities in Palacios. The XED decoder cannot be used when building Palacios as a Linux kernel module.

It is important to note that this code is separate from the main source tree of Palacios and is available under separate licenses. We include it only for convenience. You can install your own copies of these tools from their respective web sites, build them, and straightforwardly configure Palacios to use them. They are also optional: Palacios will run the guest boot environment you point it to at configuration time, and its internal instruction decoder is sufficient for virtualization purposes.

3.1 Getting and installing a source tarball

Simply download the Palacios release tarball from `http://www.v3vee.org/download`. When you untar it into your working directory, a subdirectory called `palacios` will be created. You will find all of the Palacios source code n this directory.

3.2 Using the git repository

We highly recommend using git to obtain a copy of Palacios, even if you only want to take a quick look at the codebase. Using git, you will not only get access to the code, but also be able to readily fetch updates and browse the development history of Palacios.

Get git The first thing you need to do is install git on your local system. You can download the appropriate binaries or source at `http://www.git-scm.com/download`. On Linux, you can usually install git using a package manager.

For example, on Ubuntu or Debian:

```
sudo apt-get install git
```

On RedHat or Fedora:

```
sudo yum install git
```

Once git is working, you 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"
```

Cloning our repository Having installed git, you can now clone from our repository:

```
git clone http://v3vee.org/palacios/palacios.web/palacios.git
```

This will checkout the head of the master branch into the subdirectory `palacios`. You probably want the devel branch or one of the release branches. To switch to the devel branch, simply execute:

```
cd palacios
git checkout --track -b devel origin/devel
```

To checkout the 1.3 branch:

```
git checkout --track -b Release-1.3 origin/Release-1.3
```

Information regarding other releases and branches is available on the website.

A bit about git git provides for distributed development. By cloning our repository, as described earlier, you have created a local repository for your own work in Palacios based on our devel or release 1.3 branch. You can manage that repository as you will; this includes distributing access to others. You can also incorporate changes from our repository. Finally, git can package up your changes so you can send them back to us for potential inclusion into the main repository.

Although a detailed explanation of git is beyond the scope of this report, there are a few commands that you will find most useful:

- `git branch` shows the current branch you are working on.
- `git checkout <branch-name>` changes to another development branch.

- `git add <file>` adds a file to the index to be committed.
- `git commit` writes your changes to your local clone of the Palacios repository. Use the `-a` switch to commit all currently tracked files.
- `git pull` fetches changes from your origin repository and applies them to your current branch (This is like a cvs/svn update). Note that in git language, `git pull` is the equivalent of a `git fetch` followed by a `git merge`
- `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.

There are a few common use cases that will likely arise while developing Palacios with git. For example, if you have made some commits for a new feature, you might want to send a patch to one of the maintainers. First browse which commits you would like to include with, e.g.

```
git log --pretty=oneline
```

Then create a patch:

```
git format-patch <LAST_LOCAL_COMMIT_ID>
```

This will take all local commits you have made since the commit with that ID and put them in a single file that you can then e-mail to one of the maintainers, your friends, etc. The file that is created is called a "patch". You may also receive a patch. You can then apply it to your own checked out codebase using

```
git apply <PATCH_FILE>
```

If you've made a pull or a commit that you would like to revert:

```
git reflog
git reset --hard <COMMIT_ID>
```

The `reflog` command shows a chronology of your current branch. When you reset, replace `<COMMIT_ID>` with the ID of the listing you would like to revert to.

Sometimes it is convenient to pull changes from upstream without confusing the logical order of your local commits. This will also eliminate unnecessary merges in the commit stream. A common scenario here is that you're working on something that is not ready to be committed yet and you need to do a pull from the origin repository to get a new commit that someone else has made. A good way of doing this is:

```
git stash
git pull --rebase
git stash pop
```

The `stash` command will push your local uncommitted changes onto a local stack. The rebase option tells the pull command to pull from upstream but take out your local commits and stack them on top of those gained from the pull. You can then pop the stash stack to regain your uncommitted changes in your working directory.

If you would like to learn more about git, we recommend reading one of the many reference manuals listed at `http://www.git-scm.com/documentation`.

4 Building Palacios

We now describe how to configure and compile Palacios, with the example target being Palacios built as a kernel module and integrated with a Linux host OS. The output of this process is Palacios as a static library, the Palacios kernel module, and userspace tools for using the module. You can configure Palacios for other host OS targets. A common denominator for all the targets is the static library that the build process produces. This static library is suitable for linking, either directly or indirectly, with the host OS that you are using.

4.1 Configuration

Before you compile Palacios, you will need to do some manual configuration. Palacios currently uses the Kbuild tool for this process. To configure compilation options, in the `palacios/` directory, run `make xconfig` if you are using X or `make menuconfig` otherwise. If you have recently pulled down a commit or made changes that modify the configuration process, you will be prompted to make the relevant configuration decisions the next time you run `make`

The Palacios configuration is a hierarchical namespace where the leaves are individual options to enable/disable or to set to some value. Options can be dependent, the value of one option affecting the value of another. Additionally, the existence of an option can depend on the values of other options. In the following, we describe all of the current options. You can see all the options in the `make xconfig` display by selecting Options → Show All Options.

4.1.1 Essential options

Out of the box, you will want to at least set the following options to create a Linux kernel module:

- Under Target Configuration → **Target Host OS**, select the Linux Module option.
- Target Configuration → Linux Module → **Linux Kernel Source Directory** should be set to the fully qualified path of the source tree of the host kernel for which you are building the module.
- Target Configuration → **X86 Decoder** should have the Internal Palacios Decoder selected.

4.1.2 All options

The following is a description of all options in the 1.3 release, in the order in which they appear.

Target Configuration The options under this top-level space relate to the desired host OS integration, the basic set of features the core VMM will provide, and the core VMM extensions that will be included.

- **Target Host OS** selects the host OS that we will integrate Palacios with. This determines how Palacios is built and linked to form a static library, and what additional functionality is built on top of that static library. Currently, the options are Kitten, "Old Linux" (for directly linking with Linux, which is deprecated), a Linux kernel module, Minix 3, and Other OS, which is for GeekOS and other minimal integrations.
- **Red Storm (Cray XT3/XT4)** is enabled when Palacios will run on Cray XT3/XT4 hardware, which is slightly different from commodity PC hardware.

- **AMD SVM Support** is enabled when Palacios should be able to use the AMD virtualization extensions.
- **Intel VMX Support** is enabled when Palacios should be able to use the Intel virtualization extensions.
- **Compile with Frame Pointers** and **Compile with Debug Information** are enabled to support better stack traces, and kgdb, when debugging. If you use them, it is recommended that you compile with host OS with similar options.
- **X86 Decoder** selects the instruction decoder that Palacios will use. You almost certainly want to use the Internal Palacios Decoder. If you need to be able to decode any x86 instruction, you can select the XED decoder. For linkage reasons, XED is not currently supported with the Linux kernel module target.
- **Enable VMM Telemetry Support** can be selected to have Palacios periodically output internal performance data. Under this option are additional options that can be enabled to output more data. Currently, this includes Shadow Paging.
- **Enabled Experimental Options** makes configuration options that have been labeled as experimental visible to the configuration process so that you can select them.
- **Enable Checkpointing** enables the checkpoint/restore features.
- **Supported Host OS Features** allows you to tell Palacios what host OS behavior it can rely on. Currently, there are only two options: whether the host supports aligned page allocations, and the maximum number of CPUs the host will support.
- **Host Interfaces** allows you to tell Palacios which optional host interfaces are available. Section 10 describes the required and optional interfaces. It is important to note that an optional host interface implementation is not typically part of the host OS or Palacios, but rather a part of the “glue” code written to interface the two.
- **Extensions** allows you to enable optional extensions to the core VMM. Extensions are described in detail in Section 12 and a list of the included extensions in 1.3 is given there.

Standard Library Functions Palacios makes use of a range of standard C library functions internally. As the C standard library is not typically linked into code running in kernel mode, Palacios includes its own implementations of the functions it needs. However, a typical host OS will also implement some of these functions, potentially creating name collisions. Under this configuration block, you can selectively enable and disable Palacios’s implementations of individual functions to avoid these collisions. Generally, a user can ignore this section as selecting the target host OS will automatically set these options correctly.

Virtual Paging This configuration block allows you to select whether Palacios should support shadow paging, and which shadow paging models to include. These options are currently a basic virtual TLB model and a virtual TLB model with shadow page table caching. Nested paging support is always compiled into Palacios and thus does not appear as an option.

Time Management The options here allow you to configure how the guest sees the passage of time on the processor by manipulating its TSC. **Enable Time Virtualization** turns on the feature. Then, **Hide VMM Run Cost** causes the TSC that the guest sees to not advance due to time spent in Palacios. **Fully Virtualize Guest TSC** results in every read of the TSC triggering an exit to Palacios to be handled according to fully virtualized time.

Symbiotic Functions The options under this block provide for various symbiotic virtualization capabilities (SymSpy, SymCall, and Symbiotic Modules) as described in detail in a research publication [4], as well as services built using those capabilities.

VNET This option allows you to enable the VNET/P overlay networking component of Palacios, which is described in detail in Section 7.

Debug Configuration All components of Palacios can generate debugging output (a la printf). The options under this heading allow you to selectively enable such output for elements of the core VMM, including the AMD and Intel-specific interfaces, shadow and nested paging, control register handling, interrupts, timing, I/O port access, instruction decoding and emulation, machine halt, and virtual device management. The non-core components of Palacios (e.g., virtual devices, extensions, VNET, checkpointing, etc) similarly have options for selective debug output, but those options are typically included in their individual configuration sections.

BIOS Selection The options here, for the BIOS, VGABIOS, and VMXASSIST elements, allow you to select which binaries are mapped into the guest at boot time. For each element, a path to the relevant binary is given. The BIOS binary is mapped at 0xf0000, the VGABIOS binary at 0xc0000, and VMXASSIST binary at 0xd0000 (on Intel only).

Virtual Devices Every virtual device in Palacios can be optionally enabled or disabled in this very large configuration block. More than half of the Palacios codebase consists of virtual devices. If memory usage is a concern, you can substantially trim Palacios by selecting only the virtual devices that are needed. In order for a guest to use a virtual device, that device must be enabled here, built into Palacios, and the guest configuration must specify an instance of it. More details on virtual devices and the list of currently included virtual devices can be found in Section 11.

4.2 Compilation

Once Palacios is properly configured, you can build it by running the following:

```
cd palacios/
make all
```

If you would like to see the detailed compilation steps, run `make V=1 all` instead.

The compilation process creates a static library named `libv3vee.a` in your directory. This contains Palacios. If your host OS target is the Linux kernel module, the compilation process will continue to create the module based on this library. That module will appear as `v3vee.ko`. This module can be dynamically loaded into the Linux host kernel using `insmod` Note the location of the kernel module, as you will need it later when running Palacios.

Palacios on Linux also requires that you build several tools that run in userspace in the host OS. These must be built separately from the module. To build them:

```
cd palacios/linux_usr
make
```

This builds several binary files prefixed with `v3_`. Your distribution may also include a prebuilt binary, `x0vncserver`, which is a small, statically linked, VNC server that can be used as a backend for graphics card virtual devices and the Graphics Console host interface. You will need all of these binaries later when interacting with Palacios.

5 Guest configuration

Palacios runs guest images, which are built from an XML-based configuration file and zero or more binary files using a userspace tool. The configuration file allows the user to control the available hardware in the VM and to associate it with data it will employ.

5.1 Configuration syntax and semantics

In the `palacios/utils/guest_creator` directory, you will find that we have provided a default guest configuration named `default.xml`. This file contains examples of most of the currently possible options. Some options, including experimental options, may not be documented at a given point in time. In most cases, to create your own configuration file, you can simply make a copy of this default file, and then selectively comment or uncomment elements of interest, and edit parameters. It is important to note that Palacios will attempt to create a VM with whatever configuration you specify, even if it does not make sense.

In the `palacios/vm-configs` directory, you will find several additional configuration files for common configurations. You can also visit the V3VEE web page to download other configurations and prebuilt image files.

5.1.1 Basic configuration

The basic structure of a configuration file is as follows:

```
<vm class="VM_CLASS">
 MEMORY_CONFIG
 VMM_CONFIG
 EXTENSIONS
 PAGING CONFIG
 SCHEDULING_CONFIG
 CORES_CONFIG
 MEMMAP_CONFIG
 FILES
 DEVICES
</vm>
```

Here, the VM_CLASS specifies what overall hardware model your VM will conform to. The standard option here is PC for a standard x86 PC model. Virtually all actual physical hardware conforms to this model. Other options are currently experimental.

MEMORY_CONFIG is a block describing your VM's physical memory. The syntax is

```
<memory [alignment="MEM_ALIGN"]> MEM_SIZE </memory>
```

where MEM_SIZE is the amount of physical memory in MB. The mapping of this memory into the address space conforms to the VM's class. For the PC class, the address space includes the assorted holes and conventional/extended/etc memory map. Typically, the guest BIOS will report this map via an e820 table. The optional memory alignment option allows you to force the guest's physical memory to be allocated from the host's physical memory starting at a host physical address that is an integer multiple of the MEM_ALIGN parameter. For example, if "2MB" is given, then the host physical memory block allocated to use as the guest memory will be aligned at a 2MB boundary. The purpose of this parameter is to provide large page support in the virtual paging infrastructure.

VMM_CONFIG represents blocks of VMM (as opposed to VM) configuration for this guest. The only current block is for telemetry:

```
<telemetry> "enabled"|"disabled" </telemetry>
```

If Palacios has been configured with telemetry, and the VM configuration enables it, then telemetry information will be output as the VM executes.

EXTENSIONS is a block containing a list of extensions to the core VMM that are to be enabled for this VM. If an extension has been configured and compiled into Palacios, and it is listed in a VM's configuration, then it is active while the VM runs. The overall syntax of this block is:

```
<extensions>
  <extension name="EXT_NAME" >
     EXTENSION-SPECIFIC CONFIGURATION OPTIONS
  </extension>
</extensions>
```

Here, EXT_NAME is the name of the desired extension to add. The extension-specific configuration options are outside of the scope of this discussion.

PAGING_CONFIG describes how virtual paging is to be handled for this guest, if possible given the hardware capabilities and Palacios's configuration. There are two general modes, nested paging and shadow paging. A nested paging configuration has syntax of this form:

```
<paging mode="nested">
 <large_pages> "true"|"false" </large_pages>
</paging>
```

If large_pages is set to true, then the nested paging implementation will use large pages in constructing the nested page tables. This will enhance performance. If you specify nested paging on hardware that does not support it, Palacios will automatically switch to using shadow paging with the default strategy.

The alternative, a shadow paging configuration, has syntax of this form:

```
<paging mode="shadow">
 <large_pages>"true"|"false"</large_pages>
 <strategy> SHADOW_STRATEGY </strategy>
</paging_mode>
```

Here, if `large_pages` is true, Palacios will attempt to use large pages in the shadow page tables to mirror the use of large pages in the guest page tables. This will enhance performance if possible. The `SHADOW_STRATEGY` is the implementation of shadow paging to be used. The normally used implementation is “VTLB”. “SHADOW_CACHE”, “VTLB_CACHING” , “DAV2M” and other experimental implementations may also available, but are generally not a good idea to use unless you know what you are doing. If you do not supply a shadow paging strategy, “VTLB” will be used.

`SCHEDULING_CONFIG` defines VM-specific scheduling options that Palacios should attempt to carry out in coordination with the host OS. Currently, there is only one option here:

```
<schedule_hz> HZ </schedule_hz>
```

`HZ` is the number of times per second that Palacios should attempt to yield each virtual core’s thread of execution to the host OS. Whether it is possible for Palacios to always achieve this goal of invoking the host OS’s yield() function `HZ` times per second depends on the minimum exit rate for the VM, which, in turn, typically depends on the timer interrupt rate of the host OS.

`CORES_CONFIG` is a block that defines how many virtual cores are available in the VM, and allows you to specialize behavior for each core if desired. The syntax is:

```
<cores count=NUM_CORES>
  <core>
     PER CORE CONFIG FOR CORE 0
  </core>
  <core>
     PER CORE CONFIG FOR CORE 1
  </core>
     ..
</cores>
```

Currently there are no valid per core configuration options, so tags of the form `<core />` should be used.

`MEMMAP_CONFIG` allows you to specify regions of the host physical address space to map into the guest physical address space, and at what guest physical addresses they will map. The syntax is of this form:

```
<memmap>
  <region>
    <start> START </start>
    <end> END </end>
    <host_addr> HOST_ADDR </host_addr>
  </region>
  <region>
    ...
  </region>
</memmap>
```

The parameters should be given in hex, prefaced with “0x”. This indicates that host physical addresses from `HOST_ADDR` to `HOST_ADDR` + (`END` - `START`) - 1 are to be mapped into the guest starting at guest physical address `START`. Note that explicit passthrough memory mappings like this are generally a bad idea unless you know what you are doing. Some virtual devices support “passthrough” device access, which better achieve the typical goals of using `memmap` blocks.

`FILES` represents a list of files that are to be included in your VM image when it is built from the configuration file. The purpose of including file contents is typically to support virtual storage devices, like CD ROM and hard drives. There are generally two use cases for this functionality at the present time. First, for testing purposes, it is very convenient for the VM image to be completely self-contained in a single file. For example, we often will embed an ISO file that is then used in a virtual CD ROM to boot the guest. The second use case is to support environments where there effectively is no file system on the host, such as on a node of a supercomputer. In such an environment, the self-contained image file can be readily copied directly to the memory of the node, allowing for VM launch to behave similarly to job launch. The syntax of `FILES` is

```
<files>
  <file id="FILEID1" filename="FILE1" />
  <file id="FILEID2" filename="FILE2" />
  ...
</files>
```

where `FILEID1` is the name by which the contents of the file `FILE1` can be referred to later in the configuration file.

`DEVICES` is typically the largest block within a VM configuration. It is a list of the virtual devices that will be attached to the VM. Each virtual device is an instance of a class of devices, and has a unique id. The configuration of a virtual device is specific to the class of device. In the next section, we describe the common virtual devices and their syntax. The syntax of `DEVICES` is

```
<devices>
  <device class="DEVCLASS1" id="DEVID1">
     DEVCLASS1-specific configuration
  </device>
  <device class="DEVCLASS2" id="DEVID2">
     DEVCLASS2-specific configuration
  </device>
  ...
</devices>
```

The configuration of a virtual device may refer to another virtual device by its ID, however the device referred to must appear in the configuration before the referring device.

It is possible to specify a device class which has not been built into the VMM. If this is the case, Palacios will not complete the creation of the VM or be able to launch it.

Palacios has numerous virtual device implementations that can be included in a VM using the syntax given earlier. We next describe most of these devices and their typical configuration options.

5.1.2 Basic devices

Keyboard and mouse A PS/2-compatible keyboard and mouse controller can be attached via

```
<device class="KEYBOARD" id="keyboard" />
```

There are no options.

NVRAM This device provides a PC-compatible non-volatile RAM for basic machine configuration. The NVRAM also includes the real-time clock (RTC) component of a basic PC-compatible machine. The syntax is:

```
<device class="NVRAM" id="nvram" />
   <storage> STORAGE_ID </storage>
</device>
```

Here, the optional `storage` block indicates the device that knows the configuration of hard drives, CD ROMs, and other devices that should be reported by the BIOS. The NVRAM will populate its storage-based fields using the device noted here. It's important to realize that this configuration is minimal - the NVRAM contents are really only used significantly by the BIOS during bootstrap.

5.1.3 Timing devices

Palacios supports the typical set of timing devices and registers that can be expected on a modern PC, except, currently, for the HPET.

TSC The CPU cycle counter can be selectively virtualized, as described earlier.

APIC Timer The APIC timer is part of the APIC device, described later.

PIT A PC-standard 8254 programmable interval timer device can be attached via

```
<device class="8254_PIT" id="PIT" />
```

There are no options.

RTC A PC-standard RTC is included as part of the NVRAM, described earlier.

5.1.4 Interrupt controllers / multicore functionality

Palacios supports both the legacy single processor PC interrupt logic, and the basic interrupt logic of the Intel Multiprocessor Specification.

PIC A PC-standard legacy 8259A programmable interrupt controller pair in the standard master/slave configuration can be attached via

```
<device class="8259A" id="PIC" />
```

There are no options.

LAPIC By including this device, a local advanced programmable interrupt controller is attached to every core in the system, and these LAPICs are then connected via an ICC bus. The LAPICs provide IPI functionality and interact with the virtual core threads in Palacios in order to implement the Intel Multiprocessor Specification, which is the common PC model for multiprocessors that almost all multiprocessor or multicore guest OSes rely on. The syntax is

```
<device class="LAPIC" id="apic" />
```

There are no options.

IOAPIC The I/O APIC device is the preferred way of performing device interrupt delivery on a multicore system. Unlike the PIC, which can deliver interrupts only to the first core, the I/O APIC can deliver interrupts to any core according to delivery criteria set by the guest OS. The syntax is

```
<device class="IOAPIC" id="ioapic" >
  <apic>apic</apic>
</device>
```

The required `<apic/>` parameter specifies the LAPIC device in the system. Recall that the LAPIC device represents an LAPIC on each core in the system, and an ICC bus that allows those LAPICs to communicate. Here we are stating that the IOAPIC is also attached to that ICC bus.

MPTABLE The MPTABLE device inserts a table into the guest physical memory that describes the processors and interrupt processing structure of the system. The table is dynamically created based on the core configuration, LAPIC configuration, IOAPIC configuration, and PIC configuration. This table conforms to the requirements of the Intel Multiprocessor Specification and is the way in which the boot processor finds the other processors in the VM. The syntax is

```
<device class="MPTABLE" id="mptable" />
```

There are no options.

5.1.5 Northbridge and Southbridge

These devices implement the interface of the critical elements of the system board chipset.

i440FX This device is an i440FX-compatible northbridge. The syntax to include it is

```
<device class="i440FX" id="northbridge">
    <bus>PCI_BUS</bus>
</device>
```

The bus options describe the PCI buses that are supported by the northbridge, where `PCI_BUS` should be the ID of a PCI bus that has already been described. All PC-class VMs should have a northbridge, unless you know what you're doing.

PIIX3 This device is a PIIX3-compatible southbridge. The syntax to include it is

```
<device class="PIIX3" id="southbridge">
    <bus>PCI_BUS</bus>
</device>
```

The bus options describe the PCI buses that are supported by the southbridge, where `PCI_BUS` should be the ID of a PCI bus that has already been described. It may seem odd that both northbridge and southbridge refer to a PCI bus. Both are programmable devices and thus must expose interfaces to the OS. They expose these interfaces via the PCI mechanisms (e.g., BARs), and these interfaces appear on the PCI bus that is selected. In other words, the interfaces to the northbridge and southbridge are PCI devices. All PC-class VMs should have a northbridge and southbridge, unless you know what you're doing.

5.1.6 PCI bus and passthrough/host devices

The VM can have a PCI bus. Many devices in Palacios are PCI devices and so will require the addition of a PCI bus. We also describe two special devices which allow passthrough access to physical PCI devices on the system, and to PCI virtual devices that are implemented in the host OS.

PCI The PCI device introduces a PCI bus into the guest and is needed if you will be attaching any PCI virtual device, or a passthrough device. The syntax is:

```
<device class="PCI" id="pci0" />
```

PCI_PASSTHROUGH This device allows you to map a PCI device available in the physical hardware into the guest. The syntax is:

```
<device class="PCI_PASSTHROUGH" id="MY_ID">
  <bus>PCI_BUS_ID</bus>
  <vendor_id>VENDOR_ID</vendor_id>
  <device_id>DEV_ID</device_id>
  <irq>IRQ_NUM</irq>
</device>
```

Here, `PCI_BUS_ID` is the ID of the virtual PCI bus to which this device will be added. The `VENDOR_ID` and `DEV_ID` identify the device to be mapped. Palacios will scan the physical PCI buses of the machine looking for the first device that matches on vendor and device id. This device is then mapped. Additionally, because Palacios currently cannot determine the IRQ number that the device's PCI interrupt line maps to, the user must supply this, which is the `IRQ_NUM` parameter.

It's important to note that if the PCI device does DMA, the guest OS needs to be modified so that DMA target addresses are offset by the same amount that the guest physical address space is offset with respect to the host physical address space.

PCI_FRONT This device provides a frontend to PCI host devices. A host device is one that is implemented in the host OS instead of in Palacios. Palacios access this implementation via the Host Device interface. In the Linux kernel module integration, this implementation can even be in userspace. The syntax for adding a PCI frontend device is simple:

```
<device class="PCI_FRONT" id="MY_ID" hostdev="HOST_URL">
  <bus>PCI_BUS_ID</bus>
</device>
```

Here, the `PCI_BUS_ID` is the bus to which to attach the device, while `HOST_URL` is the name of the device implementation to attempt to open via the Host Device interface.

5.1.7 Frontend and Backend Devices

Palacios implements a split device structure to enable the combination of I/O mechanisms (backends) with arbitrary emulated device interfaces (frontends). For example, this makes it possible for the IDE frontend device to emulate a CD ROM or ATA hard drive on top of different storage backends, such as files, memory, or the network. The split device structure makes it possible for backends and frontends to be developed independently.

Currently, the split device structure applies to four classes of devices:

- **Block Devices:** These devices conform to basic storage device behavior. The expectation here is that random read and write accesses will occur conforming to the capacity of the device.
- **Char Devices:** These devices assume byte level read and write behavior that conforms to a stream like abstraction.
- **Network Devices:** These devices operate on network packets, and contain interfaces for exchanging packets between the backend and frontend.
- **Console Devices:** These devices deal with console interaction and allow multiple console clients to connect to an emulated console device.

In order to add a functioning device to your configuration, you must specify both a frontend device, which is visible to the guest, and a backend device, which implements the underlying class of functionality.

First, a frontend device is specified using the following general syntax:

```
<device class=FRONTEND-CLASS id=FRONTEND-ID>
    GLOBAL FRONTEND DEVICE CONFIGURATION
</device>
```

The global frontend device configuration governs how the frontend device will connect to and interact wit the guest. For example, it might include the PCI bus to which the device is attached, or how many subsidiary buses it provides.

Second, a backend device is specified using the following general syntax:

```
<device class=BACKEND-CLASS id=BACKEND-ID>
    BACKEND DEVICE CONFIGURATION
    <frontend tag=FRONTEND-ID>
        FRONTEND DEVICE INSTANCE CONFIGURATION
    </frontend>
</device>
```

The backend device is used to determines how the underlying functionality is handled by the VMM. The backend configuration includes options for the backend device itself, the specific frontend it will be connected to, and additional options that will be passed to the frontend that specify how the frontend should emulate the backend device instance.

The frontend block is perhaps the most confusing part of the split architecture. One frontend device may make visible several underlying backend devices. For example, a storage controller may support multiple drives. Therefore, when the backend device is specified, its frontend block indicates to the frontend device how this particular backend device should appear in the guest. For example, it might specify that backend device should appear as a particular kind of drive on a particular channel of the controller.

We will now examine the different split device architectures in more detail.

5.1.8 Storage controllers and devices

A VM is given storage through a combination of frontends and backends. A storage frontend is a virtual device visible directly to the VM. A storage backend is invisible to the VM and actually implements the data storage mechanism. A frontend is added first, followed by a backend that specifies it.

IDE controller frontend This device introduces an PCI IDE controller and ATA + ATAPI device framework, which is necessary if you plan to include CD ROM drives or ATA hard drives. The syntax to include it is

```
<device class="IDE" id="ide">
    <bus>PCI_BUS</bus>
    <controller>SOUTHBRIDGE</controller>
</device>
```

where `PCI_BUS` is the ID of the bus to which to attach the controller, and `SOUTHBRIDGE` is the ID of the southbridge that governs that bus.

This device acts both as the IDE controller and as a frontend for common kinds of storage devices that can be attached to it, namely ATA hard drives and ATAPI CD ROMs. When a storage backend attaches itself to the IDE controller, it also indicates what form of storage device it is to appear to be.

Virtio block device frontend A second form of frontend for storage is the Linux virtio-compatible block device. The virtio block device has a simple syntax:

```
<device class="LNX_VIRTIO_BLK" id="blk_virtio">
     <bus>PCI_BUS</bus>
</device>
```

RAMDISK storage backend The RAMDISK backend keeps its storage in RAM. The RAM can be initially populated with the contents of a file. A common use case for this is when we want to embed a boot CD into the image file, which would look like:

```
<device class="RAMDISK" id="CD0">
  <file>FILE_ID</file>
  <frontend tag="IDE_ID">
```

```
      <model>V3Vee CDROM</model>
      <type>CDROM</type>
      <bus_num>BUS_NUM</bus_num>
      <drive_num>DRIVE_NUM</drive_num>
    </frontend>
  </device>
```

This indicates that the ramdisk is to be initially populated using the data from the file previously indicated for embedding into the image and named `FILE_ID`. The ramdisk will attach itself to the controller (here an IDE Controller we previously added) indicated by `IDE_ID` That controller is to make it appear as a CDROM attached to the indicated IDE bus at the indicated drive number (i.e., master or slave).

FILEDISK storage backend The FILEDISK backend keeps its storage in files in the host OS's filesystem. A common use case for this is to store the hard drive of a VM, for example:

```
  <device class="FILEDISK" id="ATA0" writeable="1">
   <path>/home/foo/mydisk.data</path>
   <frontend tag="IDE_ID">
      <model>V3Vee ATA Drive</model>
      <type>HD</type>
      <bus_num>BUS_NUM</bus_num>
      <drive_num>DRIVE_NUM</drive_num>
    </frontend>
  </device>
```

This indicates that the filedisk is writeable and is backed by the host path */home/foo/mydisk.data*. The filedisk will attach itself to the controller (here an IDE Controller we previously added) indicated by `IDE_ID` That controller is to make it appear as an ATA harddisk attached to the indicated IDE bus at the indicated drive number (i.e., master or slave).

NETDISK storage backend The NETDISK backend accesses its storage over the network. A common use case for this is to store the hard drive of a VM on a diskless cluster, for example:

```
  <device class="NETDISK" id="ATA1">
   <IP>129.105.49.33</IP>
   <port>9789</port>
   <tag>mydisk</tag>
   <frontend tag="IDE_ID">
      <model>V3Vee ATA Drive</model>
      <type>HD</type>
      <bus_num>BUS_NUM</bus_num>
      <drive_num>DRIVE_NUM</drive_num>
    </frontend>
  </device>
```

This indicates that Palacios should attach to a netdisk server running on 129.105.49.33, port 9789, and request access to the disk named there as “mydisk”. If it’s available, it should be attached to the VM’s IDE controller (indicated by `IDE_ID` That controller is to make it appear as an ATA harddisk attached to the indicated IDE bus at the indicated drive number (i.e., master or slave).

TMPDISK storage backend The TMPDISK backend makes available storage for the lifetime of the VM on the host. The storage is in memory, starts empty, and is discarded when the VM is stopped. A common use case for this is to store the swap drive drive of a VM on a diskless cluster, for example:

```
<device class="TMPDISK" id="ATA2">
  <size>SIZE</size>
  <frontend tag="IDE_ID">
    <model>V3Vee ATA Drive</model>
    <type>HD</type>
    <bus_num>BUS_NUM</bus_num>
    <drive_num>DRIVE_NUM</drive_num>
  </frontend>
</device>
```

This indicates that Palacios should create a temporary disk of size `SIZE` MB and attach it to the VM's IDE controller (indicated by `IDE_ID` That controller is to make it appear as an ATA harddisk attached to the indicated IDE bus at the indicated drive number (i.e., master or slave).

5.1.9 Network interfaces

Similar to storage devices, NICs also are partitioned into frontends that appear in the guest and backends that implement actual packet sending and receiving. The two primary backends are the VNET overlay and the network bridge, both of which are described in detail in Section 7. Here, we focus on the frontends.

NE2000 frontend The NE2000 is a simple, PCI-based NE2000-compatible network adapter. The syntax for adding it is simple:

```
<device class="NE2000" id="ne2000">
   <mac>MACADDR</mac>
   <bus>PCI_BUS_ID</bus>
</device>
```

Here, the `MAC_ADDR` is the hardware address of the card, while `PCI_BUS_ID` is the ID of the PCI bus to which it is attached.

RTL8139 frontend This is a simple, PCI-based RTL8139-compatible network adaptor. The syntax is the same as for the NE2000:

```
<device class="RTL8319" id="rtl8139">
   <mac>MACADDR</mac>
   <bus>PCI_BUS_ID</bus>
</device>
```

VMNET frontend A simple hypercall-based NIC is also available:

```
<device class="VMNET" id="vmnet" />
```

Virtio NIC frontend A Linux virtio-compatible NIC frontend is also available, and has the following syntax:

```
<device class="LNX_VIRTIO_NIC" id="nic_virtio">
     <bus>PCI_BUS</bus>
     <mac>MAC_ADDR</mac>
</device>
```

This device provides the guest with a frontend for low-overhead access to a network back end. This can be associated with numerous services, which are described in more detail in Section 7.

5.1.10 Serial port, virtio serial console, and stream backend

Palacios includes an 8250/16550-compatible serial port frontend device. This can be attached to a stream backend device to allow you to interact with the serial port from the host. Note that this device provides 4 serial ports ("COM1" through "COM4"). A Linux virtio console frontend device is also available.

SERIAL frontend The syntax for inserting the serial port device is straightforward:

```
<device class="SERIAL" id="serial" />
```

There are no options.

Virtio console device frontend The console char device has a simple interface.

```
<device class="LNX_VIRTIO_CONSOLE" id="cons_virtio">
     <bus>PCI_BUS</bus>
</device>
```

CHAR_STREAM backend The typical backend used for serial ports and the serial console device is the char stream device, which has a simple syntax:

```
<device class="CHAR_STREAM" id="char_stream" >
   <name>NAME</name>
   <frontend>
     <tag>TAG</tag>
   </frontend>
</device>
```

Here, `NAME` represents the name of the stream that will be supplied to the host's Stream interface to identify the stream. The `frontend` block identifies the device, such as the serial device or the virtio console device which will serve as the frontend. The `TAG` value is the ID of that frontend device.

5.1.11 Video cards and consoles

Palacios includes a text-mode video card frontend that can interface with backends that talk to userspace tools on the host, or directly to the network. It also includes a graphics video card that interacts with the Graphics Console host interface.

In addition to the combination of a video card and the keyboard device, the serial port and virtio console devices can also be used to access the guest OS.

CGA_VIDEO frontend The CGA_VIDEO device provides baseline CGA text-mode display card for the machine. The syntax is

```
<device class="CGA_VIDEO" id="cga" passthrough="enable"|"disable" />
```

If `passthrough` is set to ”enable”, the guest's manipulations of the CGA virtual device are also sent to the underlying CGA-compatible graphics hardware. CGA is a frontend device only. Nothing is displayed unless a suitable backend device is attached, two of which are described next.

The CGA_VIDEO device does not make the screen visible to the user on its own. Export of the screen is done via a text console backend, of which there are several.

TELNET_CONSOLE backend The Telnet console makes it possible to telnet to VM's CGA_VIDEO screen (and the VM's keyboard):

```
<device class="TELNET_CONSOLE" id="telnet" >
   <frontend tag="cga" />
   <port>19997</port>
</device>
```

Here the `frontend` block identifies which text-mode display card will provide the output data for the Telnet console, while the `port` block indicates the port to which the user shall telnet to access the console. The Telnet console is useful when Palacios is integrated with a host OS that can support the Socket host interface, but cannot support the Console or Graphics Console host interfaces.

CURSES_CONSOLE backend The Curses console makes the VM's CGA_VIDEO screen (and its keyboard) visible via the Console host interface. On Linux, this is used to provide access to the console via the `v3_cons` and `v3_cons_sc` userspace tools. The syntax is simply:

```
<device class="CURSES_CONSOLE" id="curses" >
   <frontend tag="cga" />
</device>
```

The `frontend` block indicates which text-mode display card will provide the output data for the Curses console.

VGA The VGA device provides a baseline VGA graphics card for the machine. The syntax is relatively simple:

```
<device class="VGA" id="vga" passthrough="enable"|"disable"
        hostframebuf="enable"|"disable" />
```

If `passthrough` is enabled, the guest's manipulations of the VGA virtual device are also sent to the underlying VGA-compatible graphics hardware. This is primarily supported to aid debugging. If `hostframebuf` is enabled, the guest's manipulations of the VGA virtual device result in drawing activity in a host frame buffer memory region via the graphics console optional host interface, which is the common output path for the VNC-based console. One of these options must be enabled in order for guest output to be seen.

Unlike the CGA_VIDEO device, the VGA device does not support backends. It uses the Graphics Console host interface directly. With the Linux integration, the implementation of this interface interacts with a userspace tool that makes the display and keyboard visible via the VNC protocol.

5.1.12 Generic "catch-all" device

The Generic device is a "catch-all" device that can be used for several purposes, including:

1. Dropping guest interactions with a physical device.
2. Spying on guest interactions with a physical device.
3. Providing a front-end for a non-PCI Host Device.

We begin with an example of using the device for (1) or (2):

```
  <device class="GENERIC" id="generic" forward="physical_device" >
    <ports>
       <start>PORT_START</start>
       <end>PORT_END</end>
       <mode>PORT_MODE</mode>
    </ports>
    <memory>
      <start>MEM_START</start>
      <end>MEM_END</end>
      <mode>MEM_MODE</mode>
    </memory>
</device>
```

A generic device can have an arbitrary number of port and memory regions, making it straightforward to produce the profile of any PC device. Here, the I/O ports in the range `PORT_START` through `PORT_END` are intercepted and handled according to `PORT_MODE` The mode "GENERIC_PASSTHROUGH" allows I/O port reads or writes to proceed on the underlying hardware, while "GENERIC_PRINT_AND_PASSTHROUGH" additionally causes the interactions with the underlying hardware (and their results) to be printed. Similarly, "GENERIC_IGNORE" and "GENERIC_PRINT_AND_IGNORE" cause the guest's attempts to be ignored, and ignored with printing. In the latter modes writes are discarded, while reads are satisfied with zeros. Printing further requires that the Generic device itself be compiled with debugging turned on. Handling guest physical memory regions is identical.

When configured as above, any "PASSTHROUGH" activity is directed to the actual physical device. To use the Generic device for (3), as a frontend for a host device, requires only the following change:

```
  <device class="GENERIC" id="generic"
          forward="host_device" url="HOST_URL">
```

Where `HOST_URL` is a string naming the relevant device implementation to the Host Device interface. In such a generic device, any "PASSTHROUGH" activity is directed to the Host Device interface instead of to the physical hardware.

5.1.13 Other devices

There are a range of other devices that can be included, but most of these are experimental.

Virtio Balloon The Linux virtio-compatible balloon device has a simple syntax:

```
    <device class="LNX_VIRTIO_BALLOON" id="balloon_virtio">
          <bus>PCI_BUS</bus>
    </device>
```

This device allows the guest to ask the VMM when to shrink or expand its memory usage.

Additional virtio-like devices There are also several additional paravirtualized devices that have interfaces that are specific to Palacios, but that conform to the overall virtio model. These are the symbiotic, symbiotic module, and VNET virtio devices. VNET is described in Section 7. The others are outside of the scope of this document.

5.2 Some hints for your configuration file

It is generally easiest to take a working configuration file and modify it for your purposes.

A Palacios image file essentially consists of a magic cookie, a directory of blobs, including their sizes and offsets, the configuration file, and finally the blobs. When Palacios creates a VM from the image, it interprets the configuration in sequential order in a single pass. Thus, it is important that any device or other entity appear *before* references to it. Palacios currently does only minimal sanity checking of the configuration.

To configure a single core PC class VM, it's sufficient to simply have a PIC. However, more recent versions of some guests (e.g., Linux) expect to see an APIC even on a single core machine. If you run a single core guest and get an invalid address exception in high (32 bit) guest physical memory, this is almost certainly because your guest is blindly assuming an APIC to be available.

To configure a typical multicore PC class VM, it is necessary to specify multiple cores and also to include APIC, IOAPIC, and MPTABLE devices. These devices provide compatibility with the Intel Multiprocessor Specification that is needed for a typical multicore guest OS to find and boot the additional (AP) processors, and to allow all the processors to communicate via IPIs. These three devices go together and work hand-in-hand with the multiple virtual cores. You will also need to configure a PIC, however, since at least the BIOS expects to be able to use it. The following idiom is common:

```
  <cores count=NUM_CORES>
    <core />
  </cores>

...
      <device class="8259A" id="pic" />
      <device class="LAPIC" id="apic" />

      <device class="IOAPIC" id="ioapic" >
         <apic>apic</apic>
      </device>

      <device class="MPTABLE" id="mptable" />
```

Having some sort of access to a guest console is usually a good idea when getting started. The most common way to do this with a Linux host is add a CGA frontend device, a CURSES_CONSOLE backend device, and a keyboard device:

```
    <device class="KEYBOARD" id="keyboard" />

    <device class="CGA_VIDEO" id="cga" passthrough="disable" />

    <device class="CURSES_CONSOLE" id="curses" >
       <frontend tag="cga" />
    </device>
```

This combination will let you attach to the text-mode console of the guest using a simple userspace tool, allowing you to switch the screen (or current terminal window) between interacting with the host and interacting with the guest.

When you're getting started, it's usually easiest to use a bootable CD as your guest, and to build it into your image file as a RAMDISK whose initial content is supplied from a file. Thus, the following idiom is common:

```
 <files>
     <file id="boot-cd" filename="/local/path/to/my_guest.iso"/>
 </files>

...

   <device class="IDE" id="ide">
      <bus>pci</bus>
      <controller>southbridge</controller>
    </device>

   <device class="RAMDISK" id="CD0">
      <file>boot-cd</file>
      <frontend tag="ide">
        <model>V3Vee CDROM</model>
        <type>CDROM</type>
        <bus_num>0</bus_num>
        <drive_num>0</drive_num>
        </frontend>
    </device>
```

If you have a particularly large bootable CD, or a large hard drive, you will probably not want to embed this content into the image file for performance reasons. Furthermore, Linux (and other host OSes) also put limits on how large a Palacios image file can be, which in turn will limit the size of drive contents possible using the above idiom. If your guest is particularly large, you may want to use a file disk instead, replacing the ramdisk block in the above idiom with

```
 <device class"FILEDISK" id="CD0">
    <path>/path/on/host/to/my_guest.iso</path>
    <frontend tag="ide">
        <model>V3Vee CDROM</model>
        <type>CDROM</type>
        <bus_num>0</bus_num>
        <drive_num>0</drive_num>
    </frontend>
 </device>
```

Here, the ISO file will need to appear in the *host* filesystem at the indicated path.

5.3 Building the guest

Once the guest XML file has been configured, you must build a Palacios-specific image file from it and any files it references.

The tool used for creating image files `build_vm`, which itself is built in the following manner:

```
cd palacios/utils/guest_creator
make
```

You then use `build_vm` in the following way:

```
./build_vm /path/to/my_config.xml -o /path/to/my_guest_image.img
```

The output of this command is a self contained guest image that can be loaded directly into Palacios.

6 Running Palacios and its VMs

We now describe how to use Palacios to run a guest within a Linux host OS. Our description in Section 6.1 through 6.2 assumes that you already have a functioning Linux host system with access to a root shell prompt, and that you have copied the `v3vee.ko` kernel module and the assorted `v3_*` userspace tools to an accessible filesystem.

In Section 6.3 we describe how you can build a tiny Linux host environment, which is very suitable for testing Palacios under QEMU, while in Sections 6.6 and 6.7 we describe debugging using QEMU/GDB and QEMU/Linux/KGDB.

6.1 Setting up Palacios

Once you have established your environment on the host OS, you must first insert the v3vee module into the kernel:

```
insmod /path/to/v3vee.ko
```

After this step, you will want to run `dmesg` to see if Palacios reported any errors while loading and configuring the machine. Palacios should have detected all the cores of the machine and initialized the appropriate hardware virtualization support on each one. You should also see `/dev/v3vee` at this point. This is the device through which Palacios is controlled.

Next, you will want to be sure that the `v3_*` utilities are on your path:

```
export PATH=$PATH:/path/to/v3_utils
```

Finally, you will want to allocate memory for Palacios to use to support large physical memory allocations:

```
v3_mem MEM_MB
```

Here, `MEM_MB` is the number of megabytes of physical memory that should be reserved. This memory is removed from the exclusive control of Linux and placed under the exclusive control of Palacios. The memory allocation should succeed. `dmesg` will show the specifics of allocation.

Logging debugging output You may find it helpful to record debugging output from Palacios. A simple way to do this is:

```
cat /proc/kmsg > LOGFILE &
echo LOGLEVEL > /proc/sys/kernel/printk
```

6.2 Managing a VM

Creating a VM To create a new VM, execute

```
v3_create /path/to/my_guest_image.img MyGuestName
```

This creates a new VM with the appropriate configuration options and the name `MyGuestName`. If there is a creation error, examining the kernel log (e.g., run `dmesg`) will provide more details on why it failed. A created VM is not yet running.

The new VM is represented as a device and if you run `ls -l /dev/v3-*` you should see something like:

```
/dev/v3-vm0
...
```

Each VM appears as a separate, numbered device file. This device file is the control point for the VM, and is referred to by the other `v3_*` utilities.

Launching a VM To launch the newly created VM, we use:

```
v3_launch /dev/v3-vm<N>
```

where `<N>` is replaced by the number corresponding to the VM—in this case you would use `v3-vm0` This command will actually boot the VM. Now if you run `ps` you should see a kernel thread in the output that looks like

```
[MyGuestName-0]
```

This kernel thread is the first virtual core of the VM. Additional cores appear as other kernel threads, with core numbers given as suffixes. So, if `MyGuestName` were a four core VM, `ps` would show:

```
[MyGuestName-0]
[MyGuestName-1]
[MyGuestName-2]
[MyGuestName-3]
```

Interacting with the VM's console If the guest includes a `CURSES_CONSOLE` console backend, you should now be able to connect to the guest console by running

```
v3_cons /dev/v3-vm<N>
```

You will automatically be connected to the console of the guest, but to exit you can press the `Esc` key. `v3_cons` will only work correctly if you are directly connected to the console of the physical host machine. It uses raw terminal access to allow full passthrough of PC scancodes from the physical keyboard to the virtual keyboard. If you are using a virtual terminal (xterm, etc), you will want to use this instead:

```
v3_cons_sc /dev/v3-vm<N> 2> debug.log
```

`v3_cons_sc` uses cooked terminal access and converts from this ASCII character stream to scan codes. While this makes it compatible with any terminal, it also means that not all scan codes can be generated.

If the guest uses a `VGA` card, and it is configured with the host frame buffer enabled, you can connect to it by first starting a special VNC server that is among the utilities:

```
x0vncserver --PasswordFile=~/vncpasswd -rfbport 5951 /dev/v3-vm<N> &
```

This makes the VGA card (and keyboard/mouse) visible at port 5951 via the VNC protocol. If this is successfully running, you can now connect to this VM using any VNC client using a syntax like:

```
vncviewer localhost:5951
```

If the guest uses a console, such as the virtio console or a serial console, that is backed by a `CHAR_STREAM` device, you can attach to that console using

```
v3_stream /dev/v3-vm<N> SERIAL_TAG
```

where `SERIAL_TAG` is the ID of the relevant device. If you only need to see output from the console, you can use `v3_monitor` instead of `v3_stream`.

Pausing and continuing a VM To pause a VM's execution, use

```
v3_pause /dev/v3-vm<N>
```

and to resume execution of a paused VM, use

```
v3_continue /dev/v3-vm<N>
```

Migrating a VM's virtual cores A VM's virtual core to physical core mapping does not change except under explicit requests. These requests are made in the following manner:

```
v3_core_move /dev/v3-vm<N> VCORE-ID PCORE-ID
```

where `VCORE-ID` is the virtual core number to move, and `PCORE-ID` is the physical core number to move it to.

Saving and loading a VM Saving and loading a VM forms the basis for basic checkpoint/restore functionality and basic migration functionality. When a VM is saved, it is written to a store in such a way that it can be restored from it and pick up where it left off.

To save a VM, it must first be paused, and then it can be saved:

```
v3_pause /dev/v3-vm<N>
v3_save /dev/v3-vm<N> STORE URL
```

Here, `STORE` indicates the storage engine, and the `URL` indicates how the storage engine is to save the data. To restore a saved VM, we first create it, as described above, then load the saved state into it, and finally we launch it:

```
v3_create /path/to/my_guest_image.img MyGuestName
v3_load /dev/v3-vm<N> STORE URL
v3_launch /dev/v3-vm<N>
```

Stopping and freeing VMs To turn off a VM, execute

```
v3_stop /dev/v3-vm<N>
```

Once the VM is stopped, you can free its resources:

```
v3_free <N>
```

Typical guest output When launching a guest, you will see the following in succession on the console:

1. The Palacios setup process
2. The boot of the guest, starting with its BIOS, and leading through its normal boot sequence.

Palacios dumps debugging output via host OS facilities. In Linux, this is via the syslog framework. The amount and nature of Palacios debugging information that appears depends on compile-time options.

6.3 A minimal Linux host

If you're doing intensive development in Palacios, especially when using QEMU, you may find it very handy to create a minimal Linux host OS image. This will make it possible to boot your test machine very quickly.

You can compile a Linux kernel into a bootable ISO image by running

```
cd hostlinux-src-dir
make isoimage
```

This will create a file called `image.iso` in the directory `arch/x86/boot/`. If you set the following options in your host Linux configuration:

- General Setup → Initial RAM filesystem and RAM disk (initramfs/initrd) support — set this to enabled
- General Setup → Initial RAM filesystem and RAM disk (initramfs/initrd) support → Initramfs source files — set this to the directory that will become the root directory of the initial ram disk.

then the noted directory tree will become part `image.iso`. When `image.iso` is booted, the loader will populate a ramdisk with its contents, and make this ramdisk the root file system for the kernel. The upshot is that with the right initramfs source directory contents (for example, a busybox distribution), your ISO image will quickly boot your own host kernel to a root command prompt.

It is important that when the kernel boots, it be given the `movablecore` option, specifying the upper limit on the number of physical pages that can be provided to Palacios.

To test Palacios, your initramfs source files should include all of the Palacios userspace utilities (e.g., `v3_*` and possibly `x0vncserver`) described in Section 4.2, the Palacios `v3vee.ko` kernel module, and any guest images you may want to use. It is often convenient to place the appropriate `v3` commands in an init script in the guest initramfs. This obviates the need to interact with the guest and helps with rapid development and debugging.

Some test environments (e.g. PXE boot) require that the kernel image and initrd are directly available, or can run even faster than with an ISO if they are (e.g., QEMU). Avoiding the ISO also allows you to decouple building the host kernel and the initrd. To build your host in this manner, compile your host kernel in this manner:

```
cd hostlinux-src-dir
make bzImage
```

Instead of an ISO file, it creates a file called `bzImage` in the `arch/x86/boot/` directory. This is the kernel itself. Now if you modify your initramfs, you can rebuild it independently of the host kernel. Linux provides a script called `gen_initramfs_list.sh` in the `hostlinux-src-dir/scripts/` directory that will create a cpio packed archive from your initramfs. To rebuild the initramfs, run:

```
cd hostlinux-src-dir/scripts
./gen_initramfs_list.sh -l /path/to/initrd/initramfs > \
    /path/to/initrd/.initramfs_data.cpio.d
./gen_initramfs_list.sh -o /path/to/initrd/initramfs_data.cpio \
    -u 0 -g 0 /path/to/initrd/initramfs
```

We find it convenient to put these commands into a wrapper script called `make_init.sh`.

The `initramfs_data.cpio` file contains the ramdisk contents that will become the root file system, while `bzImage` contains the kernel itself.

Hints We highly recommend starting from a kernel configuration and initramfs that has is known to be working for Palacios work. Failing that, we recommend fetching a kernel directly from `kernel.org` and building an initramfs around BusyBox. This is also the configuration we recommend if you decide to build a test guest (Section 6.8).

An initramfs will probably need to contain one or more device files. Creating these will require root privileges. For example, you may need to execute

```
sudo mknod initrd/initramfs/dev/console c 5 1
sudo chown 0600 initrd/initramfs/dev/console
```

in order to create a console device in your initramfs. If you do not have root privileges, there is an alternative. Assuming that you have a directory `initrd` in which there is an `initramfs` subdirectory that contains the contents for the `initramfs_data.cpio` you are building, and your Linux configuration points to the `initrd` directory, you can then create a file `initrd/root_files` that contains directives for the creation of special files as part of the build process. The equivalent entry to the above would be

```
nod /dev/console 0600 0 0 c 5 1
```

6.4 Testing in QEMU

There are three ways to test in QEMU. The first is to boot and install your distribution within a QEMU emulated machine, copy the Palacios components and VM images to it, and then execute Palacios in the context of your distribution. The primary weakness of this is that full distributions may run very slowly under emulation. The other two ways to test with QEMU involve using a minimal host, which can be generated as described above.

One can simply boot the ISO image that resulted from the `make isoimage` host OS build command. This method is simple and intuitive, but requires that you rebuild the kernel and ISO image whenever either the host kernel or the initrd contents change. To boot with this method, run the following:

```
qemu-system-x86_64 -smp 1 -m 2047 \
-serial file:./serial.out \
-cdrom hostlinux-src-dir/arch/x86/boot/image.iso
```

Here the `-smp` switch specifies the number of processors to emulate and the `-m` switch specifies the amount of memory. You can change these according to your needs. the `-serial` option tells QEMU to write out the output from the serial port to a file called `serial.out`. You can also direct this to the terminal by using `-serial stdio`. Some users experience problems with a mismapped keyboard when running QEMU. Often, this is fixed by providing the option `-k en-us`.

QEMU has special support for booting Linux kernels. It can act as a boot loader that immediately maps in the kernel and an initrd image. If you have built these, as described in the previous section, you can boot the host OS from QEMU using a command like:

```
qemu-system-x86_64 -smp -1 -m 2047 -serial stdio -k en-us \
    -kernel hostlinux-src-dir/arch/x86/boot/bzImage \
    -initrd /path/to/initrd/initramfs_data.cpio \
    -append "movablecore=262144 debug earlyprintk=ttyS0, keep console=tty0"
```

Here, the `-kernel` option indicates the host kernel that will be booted, and the `-initrd` option indicates the initrd contents. The `-append` option is the way in which to pass additional arguments to the kernel. Here, the `movablecore` kernel option is critical as it defines the upper limit of how much memory can be handed to Palacios.

6.5 Networking in QEMU

Even when testing networking-related services, QEMU can be used. It is even possible for a user with no special privileges on the physical machine to use QEMU networking. In the following, we will use the following terms:

- *Physical machine*: the machine on which QEMU is running.
- *Physical OS*: the OS running on the physical machine.
- *Emulated machine*: the machine instance QEMU is creating.
- *Host OS*: the OS running in the emulated machine, which has Palacios embedded in it.
- *Virtual machine:* the virtual machine created by Palacios.
- *Guest OS*: the OS running in the virtual machine.

Our focus is on supporting connectivity for emulated machine, on making it appear to be connected to a network. In all cases the emulated machine appears to have ordinary networking hardware (e.g., some kind of NIC).

QEMU can be configured to provide both local and global connectivity for the emulated machine. By local connectivity, we mean allowing the emulated machine to talk to the physical machine. By global connectivity, we mean allowing he emulated machine to talk to anything that the physical machine is connected to.

Connectivity can be established using either QEMU's user networking features, or by having QEMU talk to a TAP interface, which may or may not be bridged to other interfaces. User networking does not require any special privileges, but is restrictive. TAP networking requires root privileges, but is very flexible, up to allowing you to make the emulated machine a peer on the physical machine's network.

6.5.1 TAPs and bridging

TAP (and TUN) are features of the Linux kernel (the physical OS) running on the the physical machine. A TAP is basically a virtual Ethernet NIC with the additional property that a userspace program can directly send and receive raw Ethernet packets on it. This makes it possible to directly connect userspace handling of packets with kernel handling of packets. Once a TAP interface is created, it can be manipulated as if it were a regular network interface. Additionally, userspace programs interact with the TAP interface by reading and writing a special file descriptor. For example, when a program writes to this file descriptor, the kernel will see it as if data has been received *on the wire*. If data is received on the TAP from some other source, the program can read it from the file descriptor.

An orthogonal feature of the Linux kernel is bridging. Bridging allows multiple NICs (including TAPs) to be combined into, for example, Ethernet switches. By bridging a TAP interface used jointly by QEMU and the kernel, and a regular physical NIC, it is possible for the host OS to communicate to the broader network through the physical NIC.

TUN/TAP and bridging support are components of the Linux kernel that must be specifically enabled and included in the build process for the physical OS kernel. However, most common Linux builds have them enabled by default, so you probably already have them. The following assumes you have both installed and enabled.

To enable QEMU networking with TAP, it is first necessary to create the TAP interfaces on the host, and, if applicable, bridge them to the physical network. An unbridged TAP interface (with correct routes) will enabled local connectivity. A TAP interface bridged with a physical NIC (again with appropriate routes) will allow global connectivity.

Development machines will typically have several TAP interfaces, usually with each being allocated for one developer.

Simple TAP interface for local connectivity To create and manage new TAP interfaces, you can use the `tunctl` command. If your distribution does not come with `tunctl` you can get it in Ubuntu by installing the User Mode Linux utilities package:

```
sudo apt-get install uml-utilities
```

In Redhat, Fedora, and CentOS:

```
sudo yum install tunctl
```

Even without `tunctl` you can manage and create TAP interfaces with a simple C program that provides similar functionality. To see a sample C program go to `http://backreference.org/2010/03/26/tuntap-interface-tutorial/`. We use a program similar to the one on this site called `tap_create`. To create a new tap interface with `tap_create` run the following:

```
sudo tap_create tap<N>
```

or

```
sudo tunctl -t tap<N>
```

Where `<N>` is the tap device number you would like to assign. If this ran correctly, assuming you used device number 0, if you run `ifconfig -a` you should see something like this in the list:

```
tap0      Link encap:Ethernet  HWaddr 0e:e1:90:c0:fd:d8
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:500
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Now we must bring up the new TAP interface so that QEMU can use it:

```
sudo ifconfig tap<N> up 172.2<N>.0.1 up
```

This brings up interface `tap<N>` with the IP address `172.2<N>.0.1`. It also creates a routing rule that instructs the machine to direct traffic on `172.2<N>.0.1/16` to `tap<N>`. This way if the host OS running under QEMU uses an IP address within this range, you can connect to it from the local machine. For example, if we set up `tap0` this way, if you ping `172.20.0.2`, the packet (and in general, all packets with a destination address within this range) will be directed through the `tap0` interface. The source address will appear as `172.20.0.1`. QEMU can pick up these packets in its interface and the appropriate application in the emulated machine can respond to `172.20.0.1`.

Bridged TAP interface for global connectivity If you want global connectivity, the process is as follows. We assume here that you have already installed the TAP utilities, either `tunctl` or `tap_create` as described above. You also need to make sure that you have the `brctl` utility installed. If you don't, you can get it in Ubuntu with:

```
sudo apt-get install bridge-utils
```

and in RedHat, Fedora, and CentOS:

```
yum install bridge-utils
```

First, you must bring up the physical interface, say `eth1`, with the appropriate address for your network:

```
sudo ifconfig eth1 <addr> netmask <netmask>
```

The interface may already be up, of course. Next, create the TAP interface, if it doesn't already exist, and then bring it up *without* an IP address:

```
sudo ifconfig tap<N> up
```

Then you must create a bridge device:

```
sudo brctl addbr br0
```

and add both `eth1` and `tap<N>` to it:

```
sudo brctl addif br0 tap<N>
sudo brctl addif br0 eth1
```

Finally, you need to bring up the bridge interface:

```
sudo ifconfig br0 up
```

What this has done is "plugged" both `eth1` and `tap<N>` into a virtual Ethernet switch so that they can communicate with each other.

Note that for this to work, the IP address used by the host OS in the emulated machine must be set correctly. In this case, it must be set correctly to reflect the network that `eth1` is connected to. That is, the host OS must put its NIC on the same subnet as `eth1` is on.

TAP devices and bridging do not survive a reboot, therefore it is handy to do this work in a file that init runs during startup. We often do this in `/etc/rc.local`, for example.

Making QEMU use your TAP interface After your TAP device has been created, configured, and optionally bridged, you can then have QEMU use it via a syntax similar to the following:

```
qemu-system-x86_64 -smp 1 -m 2047 \
    -serial file:./serial.out \
    -cdrom hostlinux-src-dir/arch/x86/boot/image.iso \
    -net nic,model=rtl8139 \
    -net tap,ifname=tap<N>
```

The key here is the last two lines. The first indicates that QEMU should add an emulated version of the rtl8139 NIC to the emulated machine. The second line instructs QEMU to use TAP for its networking backend for that NIC, associating it with the appropriate TAP interface device.

Once the host OS comes up and brings up the rtl8139 with a reasonable IP address (one on the subnet on which the TAP interface is), it should be possible to ping the host OS from the physical machine using that address. If the TAP has been bridged to a physical interface, it should be possible to ping the host OS from any machine that can reach the physical machine.

A good deal of this process can be automated by creating a script called `qemu-ifup` that automatically brings up the appropriate TAP interfaces and adds them to relevant network bridges if needed. You can also instruct QEMU to run this script when it starts up by providing the following option:

```
-net tap,ifname=tap<N>,script=/path/to/qemu-ifup
```

By default, QEMU will look for `/etc/qemu-ifup` and `/etc/qemu-ifdown`. If your physical OS supports sudo scripting, you can then restrict who has access to these scripts via `/etc/sudoers`. If it does not support sudo scripting, you can restrict access to the underlying utilities.

6.5.2 User networking

If you don't have root or relevant sudo access to the physical machine and physical OS you can still use networking, albeit in a more restrictive manner that is quite similar to NAT networking. The basic idea is that QEMU can create and listen to tunnel ports on the physical machine and forward the TCP connections or UDP activity on these ports to the emulated machine as Ethernet packets, and convert Ethernet packets sent by the emulated machine to activity on those connections.

With user networking, QEMU essentially includes a TCP/IP stack that it uses to implement a virtual network behind a NAT. While this is convenient in that it doesn't require privileged access to the system, there are some drawbacks. Namely, performance is relatively poor compared to TAP, and protocols other than TCP and UDP cannot be used. This means you won't be able to use utilities like ping that utilize ICMP traffic.

To illustrate the process of setting up QEMU user networking, we will use the following example. We want to run a vncserver on the host OS in the emulated machine that accepts connections on port 5901, and we want to be able to access that server from a client running on the physical machine. To do this, we run QEMU with the appropriate arguments:

```
qemu-system-x86_64 -smp 1 -m 2047 \
    -serial file:./serial.out \
    -cdrom hostlinux-src-dir/arch/x86/boot/image.iso \
    -net user  \
    -net nic,model=e1000 \
    -redir tcp:6901::5901
```

The last three lines are important for this discussion. The first of these indicates that QEMU should use its user networking backend. The next line tells QEMU to install an e1000 NIC into the emulated machine. Finally, the `-redir` option instructs QEMU to listen on TCP port 6901 on the physical machine, and redirect connections on that port to connections on port 5901 on the emulated machine. If we then connect to 6901, QEMU will generate and handle the appropriate packets on the emulated e1000, translating between connection traffic and Ethernet packets on the e1000 NIC.

Note that the host OS must have the appropriate drivers for the NIC and it must bring up the interface appropriately. How this is done depends on whether the host OS has a DHCP client. QEMU's user networking backend uses a DHCP server to allocate IP addresses (by default within the range `10.0.2.0/24`) for the emulated machine. The virtual DHCP server is located at `10.0.2.2` and a virtual DNS server will be available at `10.0.2.3`. The virtual DHCP server hands out IP addresses starting at `10.0.2.15`. If the host OS you're booting under QEMU is configured to use a DHCP client, it can simply run:

```
ifup eth0
```

Otherwise, you must configure the interface with a static IP address within the range that the virtual DHCP server allows, for example

```
ifconfig eth0 up 10.0.2.15 netmask 255.0.0.0
```

Recall that this IP address is the first in the range that the virtual DHCP server allocates for its emulated machines. The netmask reflects this range, i.e. `10.0.2.0/24`.

Now the vncserver can be started on port 5901 under the host OS in the emulated machine. And you can connect to it by connecting to localhost:6901 on the physical machine.

On a typical physical machine and physical OS, the tunnel ports will be firewalled from outside access, but it is still possible to connect to them by tunneling to them via SSH. For example, if you wanted to connect to the vnc server from a different machine, you could create an ssh tunnel like:

```
ssh -L7901:localhost:6901 you@physicalmachine
```

And then point your vnc client to localhost:7901. SSH then will tunnel your connection to port 6901 on the physical machine, where QEMU will then forward it into your emulated machine where the host OS will see it as an incoming TCP connection to port 5901, where your vnc server will waiting to accept it.

For much more information concerning QEMU networking, visit
`http://wiki.qemu.org/Documentation/Networking`.

6.6 Debugging with QEMU

It is possible to do Palacios debugging using GDB and QEMU, even if your host OS does not support KGDB. We do this by starting QEMU with these additional flags:

```
-monitor telnet:localhost:4444,server
```

QEMU will now wait for a telnet connection on port 4444 before it starts. Connect to the telnet server with the command:

```
# telnet localhost 4444
```

This will bring up a monitor prompt, in which you can run the following:

```
(qemu)  gdbserver 9999
Waiting gdb connection on port '9999'
```

At this point, you can switch to a different window and fire up GDB with the Linux ELF kernel image, located in `hostlinux-src-dir/`

```
# gdb vmlinux
```

Then set the target to the QEMU port:

```
(gdb) target remote localhost:9999
Remote debugging using localhost:9999 0x000083c0
```

You can now set breakpoints, single step through functions, and , among other things, examine and dump data. You will need to use `continue` instead of `run` since you are connecting to a running program. Once you `continue`, you will see the boot process start. In the above example, 4444 and 9999 are TCP port numbers. You can pick any port that is not already in use.

6.7 Debugging with KGDB

KGDB is a useful feature of Linux that allows detailed kernel debugging. Unlike the previous technique, KGDB must have cooperation from Linux, but it gives finer-grained information and is easier to use. You can also use it with physical hardware, via a serial port.

To set up KGDB, you first need to compile both the Palacios kernel module and Linux with debugging support. In the Linux `make xconfig` or `make menuconfig` dialogue, under the "Kernel hacking" section, make sure the following options are enabled:

```
Compile the kernel with debug info
Compile the kernel with frame pointers
KGDB: kernel debugging with remote GDB ->
    KGDB: use kgdb over the serial console
```

Under the "General Setup" section, enable

```
Prompt for development and/or incomplete code/drivers
```

and under "Device drivers", enable

```
Character devices -> Serial drivers ->
    8250/16550 and compatible serial support
Character devices -> Serial drivers ->
    Console on 8250/16550 and compatible serial port
```

The Linux kernel command-line parameters must also be changed so that it invokes KGDB on boot. You can find the default parameters in `arch/x86/boot/Makefile` Change the FDARGS variable so that it looks like this:

```
FDARGS = movablecore=262144 debug kgdb=ttyS0,
    38400 kgdboc=ttyS0, 38400 kgdbwait
```

This says that KGDB should use the serial port and it should wait for a connection from GDB.

In Palacios, make sure that under the ”Target Configuration” section

```
Compile with Frame pointers
Compile with Debug information
```

are enabled.

After you've compiled the kernel and Palacios, you can boot up in QEMU with something like this:

```
qemu-system-x86_64 -smp 1 -m 2047 -serial tcp::4444,server,nowait \
    -cdrom hostlinux-src-dir/arch/x86/boot/image.iso
```

This indicates that QEMU should make the serial port available for TCP networking on port 4444. As Linux boots, it will stop and prompt for a connection from GDB. To connect, run

```
gdb hostlinux-src-dir/vmlinux
(gdb) target remote localhost:4444
```

It should connect, and if you type `continue` the kernel will resume its boot process.

Physical hardware is quite similar, except that the target specified to GDB will be the relevant serial port.

6.8 Debugging with a test guest

Most interactions between the guest and Palacios are due to the guest OS, and thus it is in these interactions that bugs are most likely to be found. While Palacios's logging features, and the debugging tools QEMU and KGDB provide, can give considerable detail for the "VMM side" of these interactions, they do not give you the "guest side" of the interactions. It is often the case that seeing what is going on in the guest kernel, at source level, at the point an interaction goes awry can provide the insight to quickly quash a bug. Even being able to cause the guest kernel to emit additional debugging prints can often localize a failure to a piece of source code that then serves as the clearest representation of how to reproduce the bug. All this is equally true if you're trying to debug a BIOS or other early boot codebase.

A common debugging approach we use is to build a tiny Linux distribution consisting of a BusyBox-based userspace environment, and a customized Linux kernel we build ourselves. The process for doing this is virtually identical to the process described in Section 6.3 for building a small host OS environment for testing. The primary differences are that:

- You will want to configure and compile the test guest kernel so that it manifests the bug.
- You will modify the source code of the test guest kernel, for example to add whatever output you need, possibly using a specialized output device described later.
- Your initramfs will not contain any Palacios modules, guests, or userspace tools. A simple BusyBox-based environment is usually sufficient.

When you write the XML configuration for your test guest, you will want to add the internal `OS_DEBUG` and `BOCH_DEBUG` devices:

```
    <device class="BOCHS_DEBUG" id="bochs debug"></device>
    <device class="OS_DEBUG" id="os debug"></device>
```

The BOCHS debug device uses port 0x402 for informational output and 0x403 for debugging output. Any character emitted to these ports in the guest, via an OUT instruction, will be displayed in the log of the host. This is intended for output from ROM BIOS, VGA BIOS, or VMXASSIST code, or other pre-boot code that you may choose to run instead. The OS debug device uses port 0xc0c0. Any character emitted to that port by an OUT instruction will be displayed in the log of the host. The device also supports an output hypercall that the guest OS can use. For both devices, the display in the host log is line-buffered to avoid confusion.

7 VNET for bridged and overlay networking

Palacios includes the VNET/P system, which can be used essentially in two modes. In both cases, it acts as a backend to NIC front-end devices mapped in to the guest. In the first mode, it can simply bridge a front-end NIC to the underlying physical network (e.g. "bridged networking"). In the second mode, it attaches a front-end NIC to a VNET overlay network that can span many machines and sites.

In general, VNET is an overlay network system that allows for a globally controlled overlay topology while having a simple layer 2 (Ethernet) interface to VMs. VMs connected to a VNET network think they are all on the same local area network, regardless of where they are, and their IP addresses are the same regardless of their location. The VNET network may be bridged to the physical network at specific points. VNET has been under development for several years, and numerous research publications can be found on the `v3vee.org` and `virtuoso.cs.northwestern.edu` web sites, including the earliest vision paper [8].

VNET/P is an implementation of the VNET model in the Palacios VMM. It shares the same model and vision with our previously implemented VNET/U system, but unlike VNET/U, it is fully implemented inside the hypervisor and designed to achieve near native performance in the 1 Gbps and 10 Gbps switched networks common in clusters today. More details on the VNET/P can be found in [10].

7.1 Enabling VNET/P in Palacios

VNET/P must be enabled, and Palacios recompiled, before it can be used. Currently, it is disabled by default. To enable it, turn on the following options

```
VNET -> Enable Vnet in Palacios
Virtual Devices -> Enable VNET Backend Device
```

These assures that the core VNET/P components and the backend device that connects a VM's virtual Ethernet devices with VNET/P forwarding components are built and linked in.

You will also want to enable the following options in order to support network bridging:

```
Target Configuration -> Host Interfaces
       -> Host support for Raw Packet Transmission
Virtual Devices -> Enable Direct Bridge to Host network
```

The VNET overlay carries Ethernet packets encapsulated in UDP packets or TCP streams. Because Ethernet packets are used, the VNET abstraction can easily interface directly with most commodity network devices, including virtual NICs exposed by VMMs to the guest and fast paravirtualized devices (e.g., Linux virtio network devices) in guests.

You will need to enable one or more of the frontend NICs in Palacios to interface with VNET. These are: To enable virtual NICs in Palacios, turn on one of more of following:

```
Virtual Devices -> Enable Virtio Network Device
Virtual Devices -> NE2K
Virtual Devices -> RTL8139
```

The Virtio NIC implementation in Palacios is fully compatible with the virtio NIC driver in Linux kernel version 2.6.21 and later and will achieve the highest performance if your guest supports this driver.

7.2 Configuring your guest with bridged networking

The simplest way to use VNET/P is to support bridging of virtual NICs with the physical network. To do this, add one or more NICs to your guest configuration, for example, a virtio NIC:

```
<device class="LNX_VIRTIO_NIC" id="net_virtio1">
    <bus>pci0</bus>
    <mac>44:55:56:57:58:60</mac>
</device>
```

Then add a `NIC_BRIDGE` device to serve as a backend for your NIC:

```
<device class="NIC_BRIDGE" id="net_bridge">
    <frontend tag="net_virtio1" />
    <hostnic name="eth0" />
</device>
```

Notice that we supply the name of the NIC on the host to which to bridge the guest NIC.

Once your VM is running, configure the host's physical NIC to promiscuous mode. Your virtual NIC will appear on the network side-by-side with the physical NIC.

7.3 Configuring your guest with overlay networking

To use overlay networking in VNET/P, your guest needs to include one or more frontend NICs. For example, you might add a virtio NIC:

```
<device class="LNX_VIRTIO_NIC" id="net_virtio1">
    <bus>pci0</bus>
    <mac>44:55:56:57:58:60</mac>
</device>
```

After one or more virtual NICs are added to the guest, you need to connect these virtual NICs to the VNET/P network. To do that, a VNET/P backend device has to be created for each frontend virtual NIC. The backend binds the frontend NIC to VNET. For example:

```
<device class="VNET_NIC" id="vnet_nic1">
    <frontend tag="net_virtio1" />
</device>
```

With this configuration, all traffic that the virtual frontend NIC sees is delivered from or to the VNET overlay. How and where packets flow depends on the VNET topology and forwarding rules, which can be dynamically configured, as we describe next.

7.4 Configuring an overlay

VNET/P supports a dynamically configurable general overlay topology with dynamically configurable routing on a per-MAC address basis.

In VNET, an *Interface* is a virtual ethernet device in a VM, an *Edge* is a logical link (e.g. a UDP, TCP or other type of network connection) between two hosts through the physical network, and a *Forwarding Rule* is a rule specifying how VNET handles certain types of ethernet packets it has received, mainly based on the packet's source or destination MAC address and other information.

In VNET/P, an interface is identified by the MAC address of the virtual NIC it represents. An edge is identified by the IP address of the remote host it connects to. A forwarding rule is defined using the following format:

```
src-MAC dst-MAC dst-TYPE dst-ID src-TYPE [src-ID]
```

The syntax of each part in the rule is defined as follows:

```
src-MAC = not-MAC|any|MAC
dst-MAC = not-MAC|any|MAC
dst-TYPE = edge|interface
src-TYPE = edge|interface|any
dst-ID = src-ID = IP|MAC

MAC = xx:xx:xx:xx:xx:xx
IP  = xxx.xxx.xxx.xxx
```

src-MAC and *dst-MAC* are MAC qualifiers to be matched to the source and destination MACs of an ethernet packet. They comprise a standard MAC address with an optional prefixed qualifier. The qualifier ”not” indicates that all packets with a MAC other than the one specified should be matched. The ”any” qualifier matches any MAC address. *dst-TYPE* and *dst-ID* specify where VNET should deliver a packet if there is one that matches the specified rules. It could be either an interface or an edge. The MAC or IP address is given thereafter depending on the type of destination. Similarly, the *src-TYPE* and *src-ID* specify the source of the packet to be matched. This is optional; if the src-TYPE is not ”any”, the origin of the packet must be matched as well.

VNET/P maintains three core tables on each host: an interface table, a link table and a forwarding table. The control of these tables is done via an interface that appears in `/proc/vnet`. One of the goals of our research efforts, is to maintain and control these tables on all the hosts comprising a VNET network in a globally adaptive way to increase the performance of the application running in the set of VMs on the VNET network. Here, we describe only the mechanisms for the local configuration of these tables.

To add a rule to the forwarding table, run the following command in the host Linux's shell:

```
echo "add <src-MAC> <dst-MAC> <dst-TYPE> <dst-ID> <src-TYPE> [src-ID]" \
     > /proc/vnet/routes
```

To delete a rule from the table, first run:

```
cat /proc/vnet/routes
```

to get the index of the rule you are about to delete, and then type the following command to delete it.

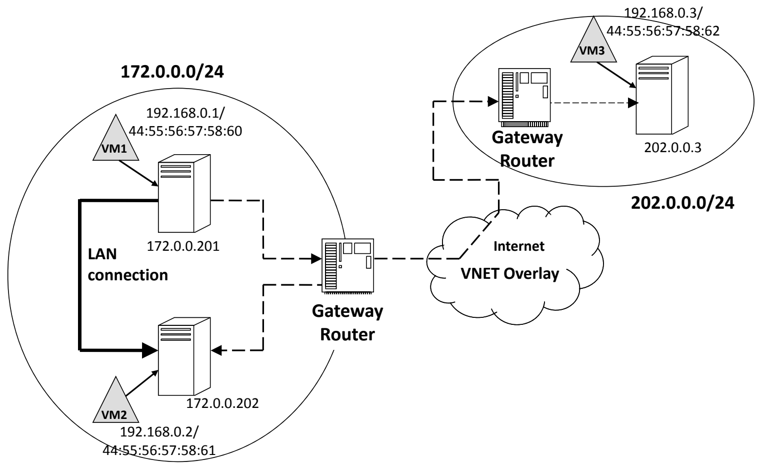

Figure 1. Network topology in the example configuration of VNET/P

```
echo "del <route-idx>" > /proc/vnet/routes
```

Similarly, to create an edge from a local host to a remote host, run following command in the host shell:

```
 echo "ADD <dst-ip> 9000 <udp|tcp>" > /proc/vnet/links
```

Where the *dst-ip* is the IP address of the remote host to be connected to. We use 9000 as the default listen port of the VNET/P daemon in the host. You can choose the connection type of the edge as either a UDP or TCP connection.

To delete an edge between a local and a remote host, first type

```
cat /proc/vnet/links
```

to get the index of the edge you are about to delete, and then run the following to delete the edge from the local VNET/P edge table.

```
echo "DEL <link-idx>" > /proc/vnet/links
```

Example global overlay configuration We now give an example on how to configure a VNET/P overlay for three VMs running on three hosts connected to each other through the IP network. All three VMs will appear to be on the same Ethernet network.

In our example configuration, 3 VMs (namely VM 1, 2, 3) are connected via VNET networking. One VM is launched on each host (as shown in Figure 1). The IP addresses of the three hosts (namely Host A, B, C) are listed below:

```
Host IP:
A: 172.0.0.201
B: 172.0.0.202
C: 202.0.0.3
```

Each guest VM is configured with one virtual NIC with MAC addresses listed as follows:

```
MAC of VMs' virtio NIC:
VM 1: 44:55:56:57:58:60
VM 2: 44:55:56:57:58:61
VM 3: 44:55:56:57:58:62
```

The VNET/P configuration steps in host A are as follows: First, we create links from host A to the other two remote hosts:

```
echo "add 172.0.0.202 9000" > /proc/vnet/links
echo "add 202.0.0.3 9000" > /proc/vnet/links
```

Then we add forwarding rules to the VNET forwarding table. Note that the forwarding entries in the example are only one way to configure all 3 VMs to be able communicate with each other.

```
echo "add not-44:55:56:57:58:60 44:55:56:57:58:60 \
      interface 44:55:56:57:58:60 any" > /proc/vnet/routes
echo "add not-44:55:56:57:58:60 FF:FF:FF:FF:FF:FF \
      interface 44:55:56:57:58:60 any" > /proc/vnet/routes
echo "add 44:55:56:57:58:60 44:55:56:57:58:61 edge 172.0.0.202 any"
echo "add 44:55:56:57:58:60 44:55:56:57:58:62 edge 202.0.0.3 any"
echo "add 44:55:56:57:58:60 FF:FF:FF:FF:FF:FF edge 172.0.0.202 any"
echo "add 44:55:56:57:58:60 FF:FF:FF:FF:FF:FF edge 202.0.0.3 any"
```

The rules for the broadcast Ethernet address support protocols such as ARP.

Similarly, the VNET/P configuration steps in host B are as follows:

```
echo "add 172.0.0.201 9000" > /proc/vnet/links
echo "add 202.0.0.3 9000" > /proc/vnet/links

echo "add not-44:55:56:57:58:60 44:55:56:57:58:60 \
      interface 44:55:56:57:58:60 any" > /proc/vnet/routes
echo "add not-44:55:56:57:58:60 FF:FF:FF:FF:FF:FF \
      interface 44:55:56:57:58:60 any" > /proc/vnet/routes
echo "add 44:55:56:57:58:60 44:55:56:57:58:61 edge 172.0.0.201 any"
echo "add 44:55:56:57:58:60 44:55:56:57:58:62 edge 202.0.0.3 any"
echo "add 44:55:56:57:58:60 FF:FF:FF:FF:FF:FF edge 172.0.0.201 any"
echo "add 44:55:56:57:58:60 FF:FF:FF:FF:FF:FF edge 202.0.0.3 any"
```

Finally, the configuration steps for host C:

```
echo "add 172.0.0.201 9000" > /proc/vnet/links
echo "add 172.0.0.202 9000" > /proc/vnet/links

echo "add not-44:55:56:57:58:60 44:55:56:57:58:60 \
     interface 44:55:56:57:58:60 any" > /proc/vnet/routes
echo "add not-44:55:56:57:58:60 FF:FF:FF:FF:FF:FF \
     interface 44:55:56:57:58:60 any" > /proc/vnet/routes
echo "add 44:55:56:57:58:60 44:55:56:57:58:61 edge 172.0.0.201 any"
echo "add 44:55:56:57:58:60 44:55:56:57:58:62 edge 172.0.0.202 any"
echo "add 44:55:56:57:58:60 FF:FF:FF:FF:FF:FF edge 172.0.0.201 any"
echo "add 44:55:56:57:58:60 FF:FF:FF:FF:FF:FF edge 172.0.0.202 any"
```

8 Code structure

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

8.1 Directory structure

The overall directory structure of Palacios is shown below:

```
palacios/
   bios/      BIOS, VGA BIOS, and VT assist code
                used to bootstrap a VM
   geekos/    geekos embedding (deprecated)
   linux_module/
              Linux-specific code used to build
              a Palacios kernel module - this code
              interfaces Linux and Palacios
   linux_usr/ userspace tools for use with the
              Palacios kernel module on Linux
   manual/    developer manual for local developers
              and others
   misc/      assorted code that doesn't fit elsewhere
   modules/   (currently unused)
   palacios/  host OS-independent core of the Palacios
              codebase - this is where most of the action is
   scripts/   (internal - ignore)
   symmods/   modules for use with symbiotic virtualization
   test/      (internal - ignore)
   utils/     assorted utilities, for building VM images,
              formatting code, analyzing Palacios output, etc.
```

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

You can find more information about Linux, Kitten, Minix 3, and GeekOS on their respective web sites. The changes needed to support Palacios can be made to be relatively minor. If you're looking for an example of a thin interface layer, we recommend looking at Kitten's Palacios support. You may also find the GeekOS embedding, in the geekos subdirectory to be useful, although it is important to point out that it is now dated and we do not expect to extend it. The core OS hooks that Palacios needs have changed only minimally since the GeekOS embedding was current, and thus it can still be illustrative. 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:

```
include/              main include directory
  config/             (internal, used for configuration process)
  devices/            virtual devices
  interfaces/         optional interfaces Palacios can use or support
  palacios/           palacios core
  vnet/               VNET high performance overlay network
  xed/                XED decoder interface (XED is optional)

lib/                  static libraries to link with palacios
                      (for example, a XED interface layer)
  i386/               32 bit libraries
  x86_64/             64 bit libraries

src/                  main src directory
  devices/            virtual devices
  extensions/         optional VMM extensions
  interfaces/         optional interfaces Palacios can use or support
  palacios/           the core VMM
    mmu/              address translation options
  vnet/               VNET high performance overlay network
```

The build process will lead to object files being generated in the various src directories.

8.2 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. The palacios/palacios/src/palacios/mmu code implements the various options for virtual memory virtualization (shadow paging, shadow paging with caching, nested paging, etc).
- 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.[5] This includes the APIC device, which is intimately tied to Palacios's support for multicore guests. Most virtual devices can be optionally selected for inclusion.
- Host OS interfaces. The required interface definition is located in palacios/palacios/include/palacios/vmm.h, particularly the `v3_os_hooks` structure. The host OS must implement all features included in this structure. Optional interfaces are defined in palacios/palacios/include/interfaces. These interfaces only need to be implemented by the host OS if Palacios has been configured to include features that depend on them. Note that both required and optional interfaces also export functions that form the programming interface that the host OS uses to request that Palacios perform actions.
- Guest bootstrap functionality. This code, located in palacios/rombios includes code that is initially mapped into a VM in order to bootstrap it. Palacios is agnostic about the bootstrap code and you can change it to suit your own purposes. Palacios begins executing a VM in the same manner (and to the same specification) as a processor reset on physical x86 hardware. For a multicore guest, the first virtual core is always elected as the Boot Processor (BP) while the other virtual cores act as Application Processors (APs) and await a startup IPI.
- Extensions. This code, located in palacios/palacios/src/extensions, contains optional components that modify or extend the functionality of the core VMM. Many of our implementations of research ideas appear here.
- VNET/P. This code implements a high performance overlay network with a layer 2 abstraction presented to guests. It allows guests to connect to a virtual Ethernet LAN that can span multiple hosts. VNET/P is compatible with the VNET/U overlay network (previously just called VNET) implemented by Northwestern in the Virtuoso cloud computing project. Because VNET/P is deeply embedded in Palacios, it is able to deliver extremely high performance over 10 gigabit Ethernet and similar speed Infiniband networks.

The Palacios build process compiles each of these components and then combines them into a single static library, `palacios/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.

If you build Palacios for Linux, you will additionally see the file `v3vee.ko` appear in the top level directory. This file is the kernel module that can be inserted into your Linux kernel using *insmod*. You will also need to separately build and copy the userspace utilities.

[5]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 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_. The Intel VMCS is in vmcs.{c,h}.

As described briefly earlier, the file palacios/palacios/include/palacios/vmm.h defines the host OS interface to Palacios. Essentially, the host OS passes Palacios a set of function pointers to functionality Palacios must have. Palacios provides a set of functions that allow for the creation, launching, and control of a VM. The optional host interfaces, defined in palacios/palacios/include/interfaces operate similarly - the host registers a structure of function pointers to implementations of functionality Palacios needs, and Palacios provides additional functions that the host can use to control the functionality (if any).

The main event 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.

9 Theory of operation

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

9.1 Perspective of guest OS

A Palacios guest sees what looks to be a physical machine with one or more processors. A multiple CPU/core VM sees an environment that is compatible with the Intel Multiprocessor Specification, but single core versions need not have APICs. The guest is booted as normal from a BIOS, and interacts directly with what appears to it to be real hardware. Palacios is not a paravirtualized VMM, but it is possible to register hypercalls for the guest to use, if desired. The common, out of the box, guest interactions 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.

9.2 Integration with host OS

Palacios does require a host OS to be embedded into, but that host OS need only provide very minimal functionality in order for Palacios to work. 32 or 64 bit host OSes are supported, although our primary targets are 64 bit host OSes, particularly Linux and Kitten. The Minix group develops the Minix 3 target. The GeekOS target was developed by us, but is unlikely to see further development.

The host OS boots the machine, and then uses the Palacios functions to create and manage guest environments. From the host OS perspective, Palacios and the guest it runs are for the most part, one or more kernel threads that use typical kernel facilities. However, there are important differences:

- Palacios is involved in interrupt processing.
- Palacios needs to have exclusive access to some of the machine's physical memory.

- Palacios may bind its threads to specific physical cores.
- Palacios can generate IPIs.
- Palacios can ask the host to vector interrupts to it.
- Palacios independently manages paging for guests.
- Palacios can provide guests with direct access to physical resources, such as memory-mapped and I/O-space–mapped devices.

How the VM appears to the end-user depends on the abstractions implemented in the host OS.

9.3 Details on the Linux embedding

The integration of Palacios with Linux is implemented in `palacios/linux_module/` with user-space tools needed to control VMs provided in `palacios/linux_usr/`.

The Linux integration is the largest and most complex of all the known Palacios/host embeddings. All required interfaces and optional interfaces are implemented here, and the userspace components include the usual range of tools needed to control VMs, interact with their consoles, etc.

The embedding has been designed to require no source code changes to the Linux kernel, and only minimal changes to its configuration. Fedora 15 is compatible with Palacios out of the box, with not additional reconfiguration necessary of the host kernel. However other off-the-shelf kernels, for example from Red Hat and Ubuntu,require a minor kernel configuration change to be compatible with Palacios.

In order function correctly Palacios requires that memory **hot-remove** and **hot-add** be enabled in the host kernel. Many kernels are configured to support hot add only. The current Linux embedding of Palacios requires that hot remove also be available. Palacios uses hot remove to acquire exclusive access to large chunks of physical memory, managing this physical memory without kernel intervention.

Currently Palacios does not support running a second VMM concurrently on the same machine. It is, however, possible to initialize Palacios on only selected cores. Using this functionality it is possible to “space share” the cores among multiple VMMs, at least as far as Palacios is concerned. With minor modifications to KVM, it is possible to operate both VMMs on one machine at the same time. The specifics for this are discussed elsewhere.

By default, when the v3vee.ko kernel module is loaded, Palacios initializes the hardware virtualization support on all cores known to the host Linux. To userspace, the Linux embedding of Palacios appears as a special device, /dev/v3vee. This device supports a range of ioctls that serve to create, launch, pause, resume, checkpoint, restore, and destroy VMs. A VMs virtual cores are statically assigned to physical cores, but a userspace tool allows the user to change this mapping at any time. Multiple virtual cores (from one or more VMs) can be multiplexed on the same physical core. The number of virtual cores in a virtual machine can exceed the number of physical cores on the actual hardware.

The Linux embedding arranges so that each core of the VM appears as a named kernel thread with the virtual core number as a suffix (e.g., “foo-1”, “foo-2”, etc). Additionally, the Linux embedding arranges so that the VM appears in the /dev hierarchy, numbered according to the order it was launched (e.g., the second VM launched appears as /dev/v3-vm1). The Linux user space tools can then interact with these per-VM devices, using ioctls to access VM-specific functionality, for example to connect text and graphics consoles, userspace backends for virtual devices and interfaces, etc.

9.4 Boot process

The host OS is responsible for booting the machine, establishing basic drivers and other simple kernel functionality, and creating at least one kernel thread to run a Palacios VM. In order to create a VM, the host OS reads a VM image file into memory and passes a pointer to it to Palacios. Palacios reads a description of the desired guest configuration from the image file and calls back into the host OS to allocate physical resources to support it. It then builds the relevant data structures to define the machine, 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, the initial processor and device state, and the initial state of the paging model that is to be used. It maps into the guest's physical address space the bootstrap code (typically a BIOS and VGA BIOS.) A typical guest VM is configured with an APIC and an MPTABLE device. The MPTABLE device creates an Intel Multiprocessor Specification-compatible MP table in the guest physical memory that describes the available cores and interrupt controllers in the VM. Palacios then launches a host OS kernel thread for each virtual core. Virtual core 0 is elected as the BP, and the other cores become APs, starting in an idle state.

Palacios then uses the SVM or VT hardware features to launch the guest's core 0, 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.

In a multicore guest, the guest OS will bootstrap the other cores (the APs) at some point in its boot process. It finds the cores and the interrupt logic interconnecting them by discovering and parsing the MP table that Palacios has written into guest memory. It boots these cores by issuing interprocessor interrupts (IPIs) to them via its APIC. Palacios handles the delivery of these interrupts, in particular using the INIT and SIPI IPI messages sent to an AP core to release that core's kernel thread from the idle state, setting its entry point, and then doing a VM entry similar to what it did for the BP core. For those unfamiliar with this process, the salient points are that an AP core's startup is virtually the same as for the BP, as described above, except that its initial CS:IP (the entry point) is determined from the SIPI message sent by the guest kernel running on the BP.

9.5 Soft state and checkpoint/migration

In order to understand some aspects of Palacios's operation, it is important to understand that while the in-memory representation of a VM is quite complex, involving numerous architecture-independent and architecture-dependent data structures, these boil down to three categories:

- VM image.
- Hard state.
- Soft state.

The bulk of the in-memory representation of a VM is soft state, which can be reconstructed from the combination of the VM image (which may itself be ephemeral in memory as long as a copy is available on storage) and the hard state.

Paging provides a good example. In the VM image is a description of the physical memory layout of the VM (similar to an e820 table), as well as directives for the paging model to be used to implement it (e.g., shadow or nested, use of large pages). The hard state includes, for example, the guest's current control register set (e.g., CR3, pointing to its current page table) and the guest physical memory contents (which include that page table). For shadow paging, the soft state includes the currently in-use shadow page table structures and other elements of the virtual TLB state. All of that soft state can be discarded at any time—the shadow paging implementation will rebuild the shadow page tables incrementally, based on the physical memory layout (from the image) and the hard state from the guest (the control registers and guest page tables stored in the guest physical memory).

Checkpointing in Palacios is built on this model. A checkpoint stores only hard state. Restoring a VM from a checkpoint is very similar to booting a VM. Just as in bootstrap, the VM image is used to construct the VM's representation in memory. Following this, the representation is patched using the information in the checkpoint. Finally, the VM's cores are started in the modes, register contents, and entry points reflected in the hard state (as compared to those reflected in the reset state as in the initial VM boot process). At this point, the VM is constructed and the hard state has been restored. As the VM executes, Palacios builds up soft state incrementally.

Migration of a VM between machines essentially looks virtually the same as a checkpoint/restore process on a single machine, except that hardware compatibility is sanity-checked on the restore side.

Migration of a virtual core from one physical core to another operates in a different manner. Outside of synchronization, the main activity is that the architecture-specific elements of the VM representation in use by the destination physical core are changed. For example, on an Intel machine, the VMCS of the migrated virtual core is loaded into the destination physical core.

9.6 VM exits and entries

The guest executes normally on a core until an exceptional condition occurs. The occurrence of an exceptional condition causes a "VM exit" on that core. 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 a guest core 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);

    update_time();

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

    (guest_context,
     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. Palacios 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 *thinks* it 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 processor context that is *actually* 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. Another way of looking at it is that the shadow context reconciles and integrates the intent of Palacios and the intent of the guest OS.

9.7 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 solve 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.

Palacios can implement this abstraction in several ways, which can be broadly categorized as nested paging and shadow paging.

9.7.1 Nested paging

One way of maintaining this two level mapping is to use a hardware feature called nested paging, which is available in the second generation of AMD processors with virtualization extensions, and recent versions of Intel processors with virtualization extensions. AMD currently calls this feature "RVI (Rapid Virtualization Indexing)" while Intel calls it "EPT (Enhanced Page Tables)". Everyone else calls it nested paging.

Nested paging directly implements the above conceptual model in hardware as two layers of page tables, one layer controlled by the guest, and the other by the VMM. Palacios supports nested paging on AMD and Intel hardware that has it. Large page support can be used to enhance nested paging performance.

9.7.2 Shadow paging

Palacios can also implement the two level mapping without hardware support using shadow paging. 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.

9.7.3 Experimental paging

The Palacios codebase also includes experimental paging approaches, which are not enabled for compilation by default. These include shadow paging with predictive prefetching of guest translations, and a paging mode that dynamically switches between shadow and nested paging based on workload characteristics.

9.8 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 it 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.

Note that hooking interrupts for passthrough or to drive a virtual device is typically not done when Palacios is embedded into a complex host OS, such as Linux. It is useful functionality for some hosts, however.

Virtual devices are generally not driven by interrupt delivery, but rather by callback functions that are visible either in the core VMM/host interface (palacios/palacios/include/palacios/vmm.h) or in the optional interfaces (palacios/palacios/include/interfaces). For example, our keyboard controller virtual device works as follows. 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 the 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 "v3_deliver_keyboard_event" 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.

9.9 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.

10 Host OS interfaces

Palacios interacts with the host OS through a core, non-optional interface, and a set of additional, optional interfaces.

10.1 Core 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)(const char * format, ...)
        __attribute__ ((format (printf, 1, 2)));

    void *(*allocate_pages)(int num_pages, unsigned int alignment);
    void (*free_pages)(void * page, int num_pages);

    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 v3_vm_info * vm, unsigned int irq);
    int (*ack_irq)(int irq);

    unsigned int (*get_cpu_khz)(void);

    void (*yield_cpu)(void);

    void *(*mutex_alloc)(void);
    void (*mutex_free)(void * mutex);
    void (*mutex_lock)(void * mutex, int must_spin);
    void (*mutex_unlock)(void * mutex);

    unsigned int (*get_cpu)(void);

    void * (*start_kernel_thread)(int (*fn)(void * arg),
                                  void * arg, char * thread_name);
    void (*interrupt_cpu)(struct v3_vm_info * vm,
                          int logical_cpu, int vector);
    void (*call_on_cpu)(int logical_cpu, void (*fn)(void * arg),
                        void * arg);
    void * (*start_thread_on_cpu)(int cpu_id,
                                  int (*fn)(void * arg),
                                  void * arg, char * thread_name);
    int (*move_thread_to_cpu)(int cpu_id,  void * thread);
};
```

The `print` function is expected to take standard `printf` argument lists.

`allocate_pages()` is expected to allocate *contiguous physical memory*, specifically `numPages` worth of 4 KB pages, and return the physical address of the memory. `free_pages()` deallocates a contiguous range of physical pages. Host physical addresses should be used. `malloc()` and `free()` should allocate kernel memory and return virtual addresses suitable for use in kernel mode. These operate on host virtual addresses.

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()` is self explanatory.

The Palacios guest execution thread will call `yield_cpu()` when the guest core does not currently require the CPU. The host OS can, of course, also preempt it, as needed.

The mutex functions should allow for the creation, use, and destruction of locks on the host OS. The locks used must be appropriate for true multiprocessor/multicore locking if Palacios is to be used on more than one core.

`get_cpu()` should return the logical CPU number on which the calling thread is currently executing. What is meant by "logical" here is that the number returned should be the one used for the relevant CPU in whatever numbering scheme is used by the host OS.

`start_kernel_thread()` is self explanatory. `start_thread_on_cpu()` should create a kernel thread and bind it to the relevant logical CPU. `move_thread_to_cpu()` should immediately bind the relevant thread to the indicated logical CPU and not return until it has done so. `interrupt_cpu()` should immediately send an IPI to the destination logical CPU with the indicated vector, while `call_on_cpu()` should interrupt the destination logical CPU and call the indicated function on it.

After filling out a `v3_os_hooks` structure, the host OS calls `Init_V3()` with a pointer to this structure, and the number of CPUs that are available. In response, Palacios will determine the hardware virtualization features of each of the CPUs, and initialize hardware virtualization support on each of them. Once it returns, the following functions can be used:

- `v3_create_vm()` Construct a VM from an in-memory image.
- `v3_load_vm()` Restore a VM from an in-memory image and a hard state checkpoint.
- `v3_start_vm()` Boot or resume a constructed or restored VM.
- `v3_pause_vm()` Suspend the execution of VM.
- `v3_continue_vm()` Resume execution of a suspended VM.
- `v3_move_vm_core()` Change the mapping of a virtual core to physical core.
- `v3_save_vm()` Checkpoint a VM's hard state.
- `v3_stop_vm()` Stop VM execution.
- `v3_free_vm()` Deallocate a stopped VM.

There is currently no interface function for VM migration.

The host OS must be able to deliver certain events to Palacios, which are delivered through the following interface functions

- `v3_deliver_keyboard_event()`
- `v3_deliver_mouse_event()`
- `v3_deliver_timer_event()`
- `v3_deliver_serial_event()`
- `v3_deliver_packet_event()`

Similar to the VM management functions, these event delivery functions are associated with individual VMs. That is, the host conceptually delivers events to a VM, not to the whole of Palacios.

As can be seen, the bare outline of the core 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 9), and in pushing necessary data to Palacios for use in virtual devices (covered in Section 11).

10.2 Optional host OS interfaces

Implementing the core host OS interface is sufficient to be able to run VMs using Palacios, but advanced functionality in Palacios relies on implementations of additional interfaces. The current set of optional interfaces is as follows:

- **File**. The File interface allows Palacios to interact with the file system of the host OS. This is used to provide file-based backends for devices such as CD ROMs and ATA hard drives.
- **Socket**. The Socket interface allows Palacios to use the network stack of the host OS, for example to create TCP connections and UDP flows. This is used to provide network-based backends for various devices.

- **Packet**. The Packet interface allows Palacios to send and receive raw network packets. This is used to support VNET and to provide bridged networking.
- **Stream**. The Stream interface allows Palacios to associate a streaming device, such as a serial port, with a host-level stream. In the Linux module implementation, allows, for example, accessing a VM's serial port from a userspace program.
- **Keyed Stream**. The Keyed Stream interface presents the abstraction of a collection of named streams to Palacios. Palacios currently only uses this abstraction for checkpointing. The Linux module has implementations of this interface that map the collection to a directory of files, a hash table of in-memory streams, and tagged data over a network connection. It also includes support for the implementation of keyed streams in userspace.
- **Console**. The Console interface interfaces with a VM's text-mode display adaptor (e.g. CGA) and keyboard to allow the host OS to display the VM console as appropriate. In the Linux module, the console interface implementation provides for a userspace utility to attach to the VM and display its console in the current terminal.
- **Graphics Console**. The Graphics Console interface interfaces with a VM's text-mode and graphics-mode display adaptor (e.g. VGA), keyboard, and mouse to allow the host OS to display the VM console as appropriate. In the Linux module, the graphics console interface implementation provides for a userspace utility to attach to a frame buffer. The utility then exports this frame buffer, keyboard, and mouse to remote clients through the VNC protocol.
- **Host Device**. The Host Device interface makes it possible to implement a virtual device in the context of the host OS instead of within Palacios. Both legacy devices and PCI devices can be supported. The host-based device is then made visible in the VM through the use of the generic virtual device or the PCI frontend virtual device. In the Linux module, the host device interface implementation also supports the construction of devices in userspace. That is, a virtual device can be implemented as a user-level Linux program as well as within the Linux kernel.
- **Inspector**. The Inspector interface allows the host OS to examine and modify the state of a running VM. Any state registered for inspection in Palacios is visible through this interface. The state is presented in the form of a hierarchical namespace. In the Linux module, this interface allows making a VM's state visible and modifiable via the sysfs hierarchy.
- **GEARS**. The Guest Examination And Revision Services is a growing interface that allows the host OS to tap into and manipulate the behavior of the guest OS and its applications. Current services include software interrupt interception, Linux system call interception, Linux system call hijacking, Linux application environment variable manipulation, and code injection into Linux guests and applications.

11 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 virtual devices. These are located `palacios/palacios/\{include,src\}/devices/`, which is also where additional virtual devices should be placed.

11.1 Virtual device interface

A virtual device includes an initialization function, such as:

```
static int mydev_init(struct v3_vm_info *vm, v3_cfg_tree_t *cfg);
```

that is registered and associated with an name like:

```
device_register("MYDEV",mydev_init);
```

Then a device of class "MYDEV" is then encountered in constructing a VM from an XML configuration file, the registered initialization function will be called, and handed a pointer to the VM being constructed, and to the subtree of the XML tree relating to that device. The initialization function is expected to then parse this subtree, construct the relevant internal state of the device instance, and finally call

```
struct vm_device *dev = v3_add_device(vm, dev_id, &dev_ops, &internal);
```

Here, `dev_id` is the ID of this instance of the device, and `internal` is the internal state of the instance. `dev_ops` is a structure of form:

```
struct v3_device_ops {
  int (*free)(void *private_data);
  int (*save)(struct v3_chkpt_ctx *ctx, void *private_data);
  int (*load)(struct v3_chkpt_ctx *ctx, void *private_data);
};
```

The `save` and `load` functions are used for checkpoint/restore of the device, while `free` is used to free the internal state. In all cases the private data pointer returns the originally registered pointer to the internal state of this device instance.

Typically, the device initialization function (e.g., `mydev_init`) will create the device state, and then configure itself to correctly interact with the guest's environment. The methods for guest interaction are discussed in detail in Section 13. 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.

11.2 Included virtual devices

The current list of included virtual devices is:

- Advanced Programmable Interrupt Controller (APIC) [based on the Intel APIC specification and the Intel Multiprocessor Specification]

- Input/Output Advanced Programmable Interrupt Controller (IOAPIC) [based on the Intel IOAPIC Specification and the Intel Multiprocessor Specification]
- MPTABLE device (used to inject MP Tables into the guest as described in the Intel Multiprocessor Specification]
- Programmable Interrupt Controller (PIC) [based on Intel 8259A]
- Programmable Interval Timer (PIT) [based on Intel 8254]
- i440FX-compatible North Bridge
- PIIX3-compatible South Bridge
- PCI Bus
- IDE bus
- ATA hard drive
- ATAPI CD ROM
- CGA (text mode only) display adapter
- VGA (text and graphics mode) display adapter
- Serial port (8250/16550)
- Virtio-compatible balloon device
- Virtio-compatible block device
- Virtio-compatible console device
- Virtio-compatible network device
- NE2000-compatible network device
- RTL8139-compatible network device
- Keyboard/Mouse [based on the PS/2 interface]
- NVRAM and RTC
- Debug port for the BOCHS BIOS
- A generic interception device to inspect I/O traffic and IRQ occurrences, as well as to allow passthrough-access to non-PCI physical devices. The generic device can also serve as a frontend for host OS-based device implementations. In the Linux module, this implementation includes support for devices implemented in userspace.
- A PCI frontend device for host OS-based device implementations. In the Linux module, this implementation includes support for devices implemented in userspace.

- A PCI passthrough device that allows for passthrough guest access to physical PCI devices.

Devices that use storage (such as the CD ROM and ATA hard drive) can implement that storage on top of memory, files, and or network connections depending on which optional host OS interfaces have been implemented.

Devices that use stream communication (such as the serial port) depend on stream functionality implemented by an optional host OS interface.

The VNET/P overlay network also involves virtual devices. These are described separately in Section 7.

12 Extensions

We now describe the Palacios extension model and currently implemented extensions.

12.1 Extension model

Although the most common way of extending Palacios is the implementation of virtual devices, the Palacios framework also attempts to simplify the addition of functionality to the core of the VMM without the need to modify the core source code directly. This is important for two reasons. First, it makes the task of the extension author easier. Second, it avoids introducing timing, synchronization, or interrupt handling bugs into the subtle code at the core of the VMM.

The interface for Palacios extensions is given in `palacios/palacios/include/palacios/vmm_extensions.h`. Extensions are typically placed into `palacios/palacios/src/extensions/`.

An extension is registered using code similar to this:

```
static struct v3_extension_impl myext_impl = {
    .name = "MYEXT",
    .init = init_myext,
    .deinit = deinit_myext,
    .core_init = coreinit_myext,
    .core_deinit = coredeinit_myext,
    .on_entry = entry_myext,
    .on_exit = exit_myext
};

register_extension(&myext_impl);
```

The extension is then used by including an extension block in the XML configuration of the VM that has the matching name ("MYEXT" here). When this is seen at VM creation time, the callback function (`init_myext`) is invoked with a pointer to the VM being constructed and the XML subtree rooted at the extension block. When a core is inited, the core initialization function is called (`coreinit_myext`). The deinit callbacks are invoked as the VM is being destroyed.

Most important for typical extensions are the callbacks for VM exit and VM entry. These callbacks are invoked with the VM architecture independent and architecture-dependent state in a safe state and with normal host OS interrupt processing on. The initialization functions provide a way to register internal extension state, which is then handed back to these callbacks. The initialization functions can also hook ports, memory, MSRs, etc, for extensions, similar to devices. Guest interaction is described in more detail in Section 13.

12.2 Current extensions

The following extensions are currently present in the codebase and can be selectively enabled in the configuration process:

- Time Virtualization. This extension includes efforts to improve time virtualization across all timing devices the guest has access to.
- TSC Virtualization. This extension virtualizes the hardware cycle counter that the guest sees.
- MTRR Virtualization. This extension allows for virtualization of the MTRR registers, which control how the CPU interacts with different regions of physical memory.
- Inspector. This extension provides the backend for the Inspector host interface, making it possible for the host to access registered VM state as a hierarchical namespace.
- Machine Check Injection Framework. This extension makes it possible to inject what to the guest appear to be hardware errors. Many different kinds of errors can be injected.
- Guest Examination and Revision Services. GEARS is a collection of extensions that implement the backend to the host OS GEARS interface. This allows for manipulation of the guest in a range of ways.

13 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 CPUID instructions, IRQs, software interrupts, I/O port operations, MSR operations, and memory operations. They can also intercept certain specific host events. Components can also inject exceptions, interrupts, virtual interrupts, and software interrupts into running guests. We will now describe these interfaces and their APIs.

13.1 CPUID hooking

CPUID hooking if useful if a Palacios component wants to handle executions of the CPUID instruction to control the view of the processor that the guest sees. This is done by calling `v3_hook_cpuid`.

```
int v3_hook_cpuid(struct v3_vm_info * vm, uint32_t cpuid,
  int (*hook_fn)(struct guest_info * info,
                                  uint32_t cpuid,
 uint32_t * eax,
                                  uint32_t * ebx,
 uint32_t * ecx,
                                  uint32_t * edx,
 void * private_data),
  void * private_data);
```

The function takes the guest context, the CPUID request to hook, the callback function to invoke, and a pointer to data hand back to the callback when it is invoked. The callback function can manipulate the register state it is handed as it pleases. Notice that the callback function is given the state of the core, not the state of the whole guest. This is because CPUID is core-specific.

A matching unhook function, `v3_unhook_cpuid()` is also provided. CPUID executions can be dynamically hooked and unhooked.

13.2 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 v3_vm_info * vm,
                uint_t irq,
                int (*handler)(struct v3_vm_info * vm,
                               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 v3_vm_info * vm, 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. Dynamic unhooking is not currently supported.

The optional GEARS extension provides the ability to hook software interrupts (e.g. INT instructions), `v3_hook_swintr()`, the fast system call instructions (e.g., SYSENTER/SYSCALL), and Linux system calls, `v3_hook_syscall()`.

13.3 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 v3_vm_info * vm, uint16_t port,
                     int (*read)(struct guest_info * core,
                                 uint16_t port, void * dst,
                                 uint_t length, void * private_data),
                     int (*write)(struct guest_info * core,
                                  uint16_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.

Setting a hook callback to NULL enables passthrough operation for that operation on the specified port. This allows a guest to have direct access to hardware IO ports on a port by port basis, as decided by the VMM.

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

13.4 Memory hooking

Memory hooking is essential for building many forms of virtual devices, in particular those that are memory-mapped. Palacios supports intercepting read and write operations to specific memory regions defined at page granularity. This is done via a call to either `v3_hook_full_mem` or `v3_hook_write_mem`

```
int v3_hook_full_mem(struct v3_vm_info * vm, uint16_t core_id,
     addr_t guest_addr_start, addr_t guest_addr_end,
     int (*read)(struct guest_info * core,
                              addr_t guest_addr, void * dst,
                              uint_t length, void * priv_data),
     int (*write)(struct guest_info * core,
                              addr_t guest_addr, void * src,
                              uint_t length, void * priv_data),
     void * priv_data);

int v3_hook_write_mem(struct v3_vm_info * vm, uint16_t core_id,
      addr_t guest_addr_start, addr_t guest_addr_end,
                 addr_t host_addr,
      int (*write)(struct guest_info * core,
                              addr_t guest_addr, void * src,
                              uint_t length, void * priv_data),
      void * priv_data);
```

`v3_hook_write_mem` captures only write operations that happen to a given region of physical memory. Read operations are executed by the hardware directly on the region without trapping into a VM. This is useful for situations where you only want to capture modification events to a particular memory region, but don't care about fully virtualizing the memory contents.

`v3_hook_full_mem` captures both read and write operations allowing components to fully virtualize a given range of memory. This is useful for implementing memory mapped I/O.

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 `host_addr` argument to `v3_hook_write_mem` is used to specify the host physical memory address of the memory being hooked. Any write operation that occurs in the hooked region will be executed using this region, and all read operations will read data directly from it. After a write operation has executed the result of that operation is passed to the write hook callback function.

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.

Memory regions can be dynamically hooked and unhooked. The unhook call is `v3_unhook_mem()`.

13.5 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 v3_vm_info * vm, uint32_t msr,
                int (*read)(struct guest_info * core,
                            uint32_t msr, struct v3_msr * dst,
                            void * priv_data),
                int (*write)(struct guest_info * core,
                             uint32_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 {
        uint64_t value;

        struct {
            uint32_t lo;
            uint32_t hi;
        } __attribute__((packed));
    } __attribute__((packed));
} __attribute__((packed));
```

Similarly to I/O hooks, `NULL` read/write callback parameters are interpreted as enabling passthrough operation. This is necessary for a subset of the architectural MSRs that are virtualized automatically by the hardware. In this case the read/write operations can execute directly on hardware virtualized copies of the MSRs.

MSRs can be dynamically hooked and unhooked. The unhook function is `v3_unhook_msr()`. Additionally, a `v3_get_msr_hook()` is available to look up these hooks.

13.6 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 v3_vm_info * vm,
       v3_host_evt_type_t event_type,
       union v3_host_event_handler cb,
       void * private_data);
```

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

```
typedef enum { HOST_KEYBOARD_EVT,
       HOST_MOUSE_EVT,
       HOST_TIMER_EVT,
       HOST_CONSOLE_EVT,
       HOST_SERIAL_EVT,
       HOST_PACKET_EVT} v3_host_evt_type_t;

union v3_host_event_handler {
    int (*keyboard_handler)(struct v3_vm_info * vm,
                            struct v3_keyboard_event * evt,
                            void * priv_data);
    int (*mouse_handler)(struct v3_vm_info * vm,
                         struct v3_mouse_event * evt,
                         void * priv_data);
    int (*timer_handler)(struct v3_vm_info * vm,
                         struct v3_timer_event * evt,
                         void * priv_data);
    int (*serial_handler)(struct v3_vm_info * vm,
                          struct v3_serial_event * evt,
                          void * priv_data);
    int (*console_handler)(struct v3_vm_info * vm,
                           struct v3_console_event * evt,
                           void * priv_data);
    int (*packet_handler)(struct v3_vm_info * vm,
                          struct v3_packet_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;
};

struct v3_serial_event {
    unsigned char data[128];
    unsigned int len;
};

struct v3_console_event {
    unsigned int cmd;
};

struct v3_packet_event {
    unsigned char * pkt;
    unsigned int size;
};
```

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. The serial event contains a sequence of bytes. The console event contains a command word. The packet event contains a variable sized packet that has been received.

Currently, host events can be dynamically hooked, but not unhooked.

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

13.7 Injecting exceptions, interrupts, virtual interrupts, and software interrupts

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 * core
                       uint_t excp);
int v3_raise_exception_with_error(struct guest_info * core,
                                  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 v3_vm_info * vm, int irq);
int v3_lower_irq(struct v3_vm_info * vm, int irq);
```

It is important to note that raising an interrupt raises it on the virtual interrupt controller architecture (PIC, APIC, IOAPIC, etc) . When the interrupt is actually delivered, and to which core it is delivered, depends on the guest's current configuration of the interrupt controller architecture, for example interrupt masking, interrupt routing, interrupt flags, and the priorities of other interrupts and exceptional conditions that are also raised.

Virtual interrupts can also be raised and lowered:

```
int v3_raise_virq(struct guest_info * vm, int virq);
int v3_lower_virq(struct guest_info * vm, int virq);
```

Virtual interrupts imply dispatch directly on a specific core instead of via the interrupt controller architecture. Interrupt flags, and the priorities of the various interrupts and other exceptional conditions are observed.

Software interrupts are simply raised as they will produce immediate dispatch (just as if an INT instruction were executed):

```
int v3_raise_swintr(struct guest_info * core, uint8_t vector);
```

There is no notion of lowering a software interrupt. Priorities and interrupt flags are strictly observed, as on the actual PC hardware.

13.8 Barriers

Generally, hook callback functions, extensions, and other functionality can run independently on each core. However, synchronized activity across cores may be needed at times. To support this, Palacios provides a barrier mechanism:

```
int v3_raise_barrier(struct v3_vm_info *vm_info,
                     struct guest_info *local_core);
```

Raising a barrier on one core causes all other cores to immediately exit and invoke `v3_wait_at_barrier()` as soon as possible. Once they all have done so, `v3_raise_barrier()` will return. The calling core is now executing with exclusive access to the VM. Once the calling core finishes the tasks it needs to do, it invokes `v3_lower_barrier()`, which allows all the cores to continue execution.

14 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 unpolluted. 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.

15 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-1.3` or `devel` 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.org` for possible inclusion in the mainline codebase.

16 Acknowledgments

V3VEE is a large project, and Palacios includes contributions by many collaborators in addition to those who have authored this document.

Peter Dinda, Jack Lange, Patrick Bridges, Kevin Pedretti, and Stephen Scott are the principal investigators at Northwestern, U.Pittsburgh, U.New Mexico, Sandia National Labs, and Oak Ridge National Lab, respectively. Major Northwestern contributors to the Palacios codebase also include Lei Xia, Chang Bae, Kyle Hale, Andy Gocke, Madhav Suresh, Mat Wojcik, Brad Weinberger, Peter Kamm, Robert Deloatch, Steven Jaconette, Rumou Duan, and Yuan Tang. Major UNM contributors to the Palacios codebase also include Zheng Cui and Phil Soltero. Major Sandia contributors to the Palacios codebase also include Trammel Hudson.

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

A Our test environment

We currently use the following machines when testing Palacios:

- 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 is among the cheapest Intel processors that support VT without nested paging.
- Dell Optiplex 380 with an Intel Core 2 Duo E7550 processor, 4 GB RAM, 160 GB SATA drive, and ATAPI CD ROM.
- Dell PowerEdge R210 with an Intel Xeon X3430 processor, 8 GB RAM, 160 GB SATA drive, and ATAPI CD ROM.
- Dell PowerEdge R410 with an Intel Xeon E5620 processor, 8 GB RAM, 160 GB SATA drive, and ATAPI CD ROM.
- Dell PowerEdge T110 with a Intel Xeon X3450 processor, 4 GB RAM, 500 GB SATA drive, and ATAPI CD ROM.
- Dell Optiplex 790 with an Intel Core i7 2600 processor, 8 GB RAM, 1 TB SATA drive, and ATAPI CD ROM.
- Dell PowerEdge R415 with dual AMD Opteron 4174s, 16 GB RAM, 2 TB SAS drive, and ATAPI CD ROM.
- Gray box, dual quadcore 2.3GHz AMD 2376, 32GB memory, 500GB SATA drive, ATAPI DVD-ROM

- HP ProLiant BL460c G6 BladeSystem with dual-socket quad-core Intel Xeon X5570, 24 GB RAM, no disk
- HP ProLiant BL465c G7 BladeSystem with dual-socket 12-core AMD Opteron 6172, 32 GB RAM, no disk
- Cray XT4, specifically compute nodes with single-socket quad-core AMD Opteron 1354, 8 GB RAM, SeaStar 2.1 network interface, no disk

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

[4] LANGE, J., AND DINDA, P. Symcall: Symbiotic virtualization through vmm-to-guest upcalls. In *Proceedings of the 2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (March 2011).

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

[6] 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).

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

[8] SUNDARARAJ, A., AND DINDA, P. Towards virtual networks for virtual machine grid computing. In *Proceedings of the 3rd USENIX Virtual Machine Research And Technology Symposium (VM 2004)* (May 2004). Earlier version available as Technical Report NWU-CS-03-27, Department of Computer Science, Northwestern University.

[9] 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.

[10] XIA, L., CUI, Z., LANGE, J., TANG, Y., DINDA, P., AND BRIDGES, P. Vnet/p: Bridging the cloud and high performance computing through fast overlay networking. Tech. Rep. NWU-EECS-11-07, Department of Computer Science, Northwestern University, 2011.