

System-level Virtualization:

Overview & recent work on Loadable Hypervisor Modules

Thomas Naughton^{*†}

naughtont@ornl.gov

** Systems Research Team*

Computer Science and Mathematics Division
Oak Ridge National Laboratory

† ACET Centre

School of Systems Engineering
The University of Reading



Background

- About me
 - Oak Ridge National Laboratory since 2001, full-time staff in 2007
 - Univ. of Reading part-time/working-away since May 2007
- ORNL: System Research Team (SRT)
 - Christian Engelmann
 - Hong Ong
 - Stephen Scott
 - Anand Tikotekar
 - Geoffroy Vallée
 - * Ferrol Aderholdt (TNTech/ORAU)

Introduction

What is Virtualization?

- Mechanism for manipulating access to a resource
 - Ex. Multiplexing a single physical device between multiple users is a form of virtualization
- Virtualization vs. Abstraction
 - Objectives may differ, e.g., simplify interface

Forms of “Virtualization”

- Generic Resource Isolation Mechanisms
 - File systems: chroot() – change root directory
- Process-level Virtual Machines
 - JVM: Java Virtual Machine
 - PVM: Parallel Virtual Machine
 - LAM/MPI: Local Area Multicomputer / Message Passing Interface
- System-level Virtualization [← This talk](#)
 - Xen hypervisor
 - VMWare workstation

Why Virtualization?

- Decouple operating system / hardware
 - Server consolidation (utilization)
 - Development & Testing
- Customization of execution environment
 - Specialize for an application / user
 - Environment consistency
- Reliability
 - High availability / Fault Tolerance
 - Non-stop computing via VM migration

Virtualization Overview

Virtualization

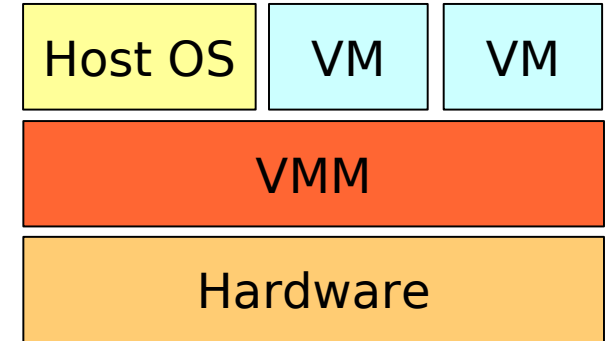
- General Terminology
 - VMM: Virtual Machine Monitor, a.k.a. *Hypervisor*
 - VM: Virtual Machine
 - Host OS: operating system run on physical machine
 - Guest OS: operating system run in virtual machine
- Classic Popek & Goldberg VMM criteria (1974)
 1. *Fidelity* – VMM environment “essentially identical” with physical machine
 2. *Efficiency* – VMM runs most operations on physical resources
 3. *Control* – VMM fully controls physical resources

Implementation: Different Approaches

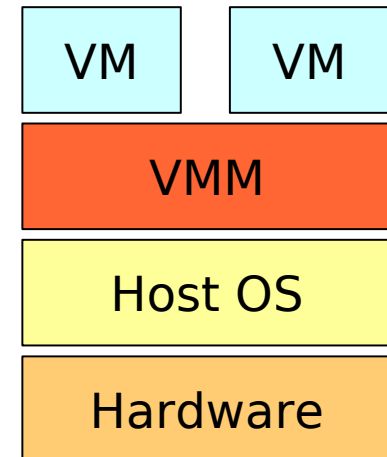
- Full-virtualization: run unmodified guest OS
 - Possibly with hardware support: Intel-VT, AMD-V
- Para-virtualization: modify guest OS
 - For performance purposes
- Emulation: different architecture for host & guest
 - Co-designed VM: dynamically translate guest ISA to native
 - e.g., Transmeta

System-level Virtualization

- First research in domain Goldberg'73
 - Type-I & Type-II
 - Distinction: VMM relative to hardware
- Xen created renewed interest
 - Performance (para-virtualization)
 - Open Source
 - Linux based
- Interests for HPC
 - System management / Customization
 - Fault-tolerance (migration)



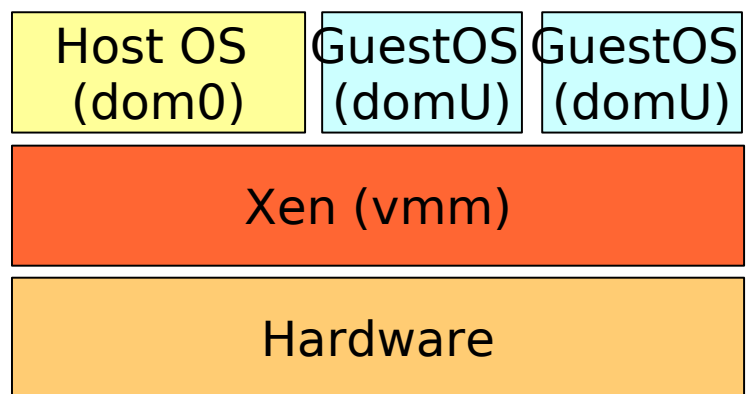
Type I Virtualization



Type II Virtualization

Xen Terminology

- Domains: term used regarding OS's running on VMM
 - User domains (domU)
 - Administrative domains (dom0)
- Hypercall: call from domain to VMM
 - Analogous to a system call (user to kernel)



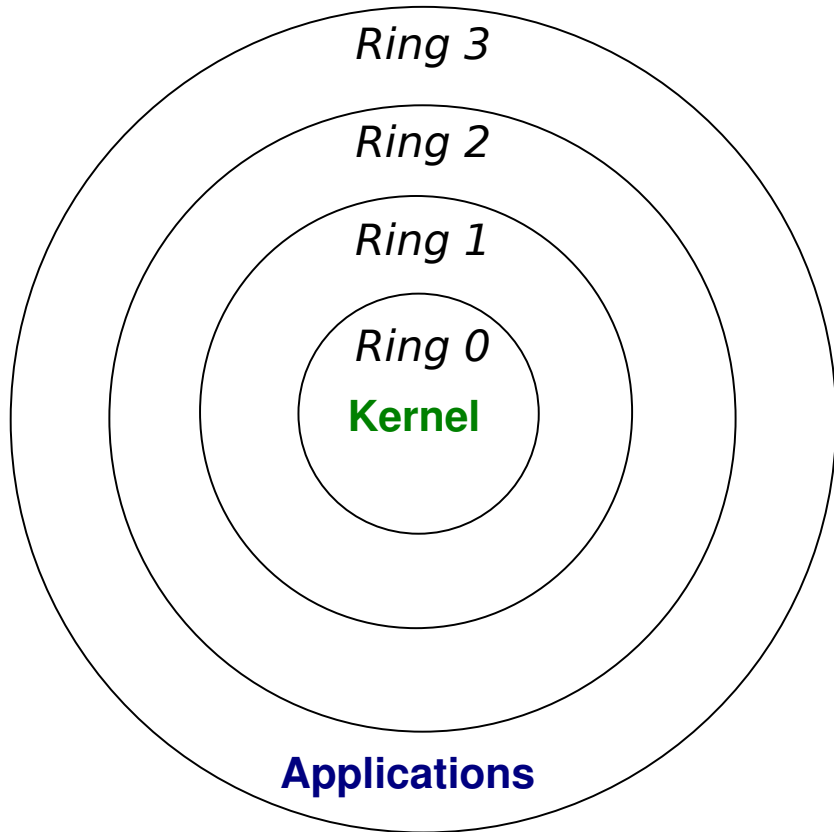
Xen: Type-I hypervisor

System-level Virtualization: Type-I vs. Type-II

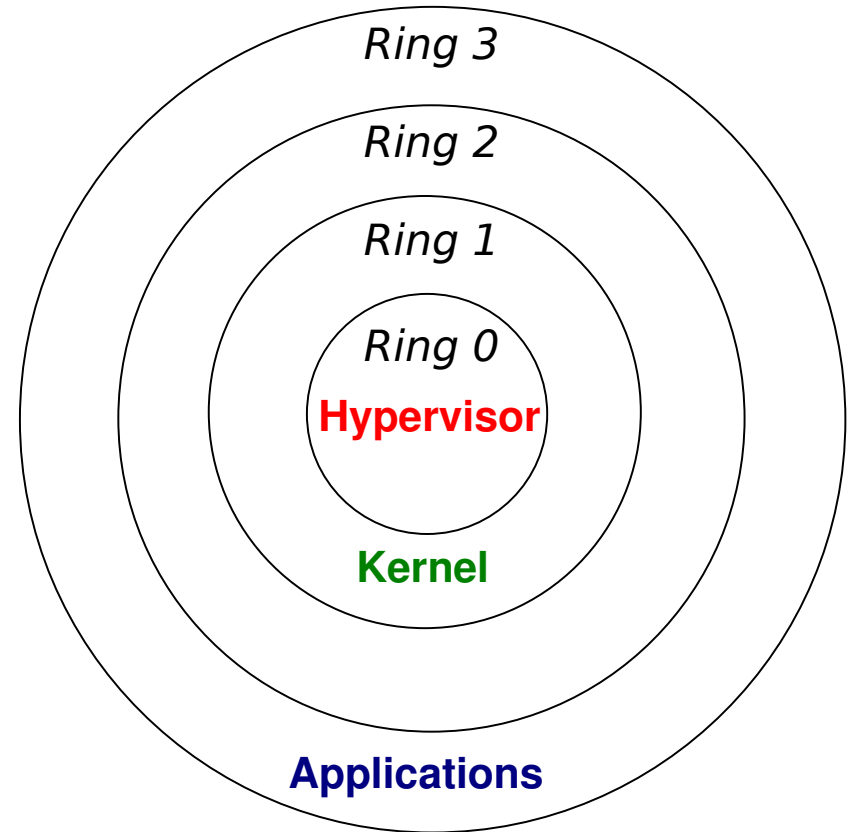
System-level Virtualization Solutions

- Several solutions
 - Xen, QEMU, VMWare, KVM, LGUEST, ...
- Basic rule of thumb
 - Type-I: performance
 - Type-II: development

Type-I: Design



x86 Architecture - Execution Rings



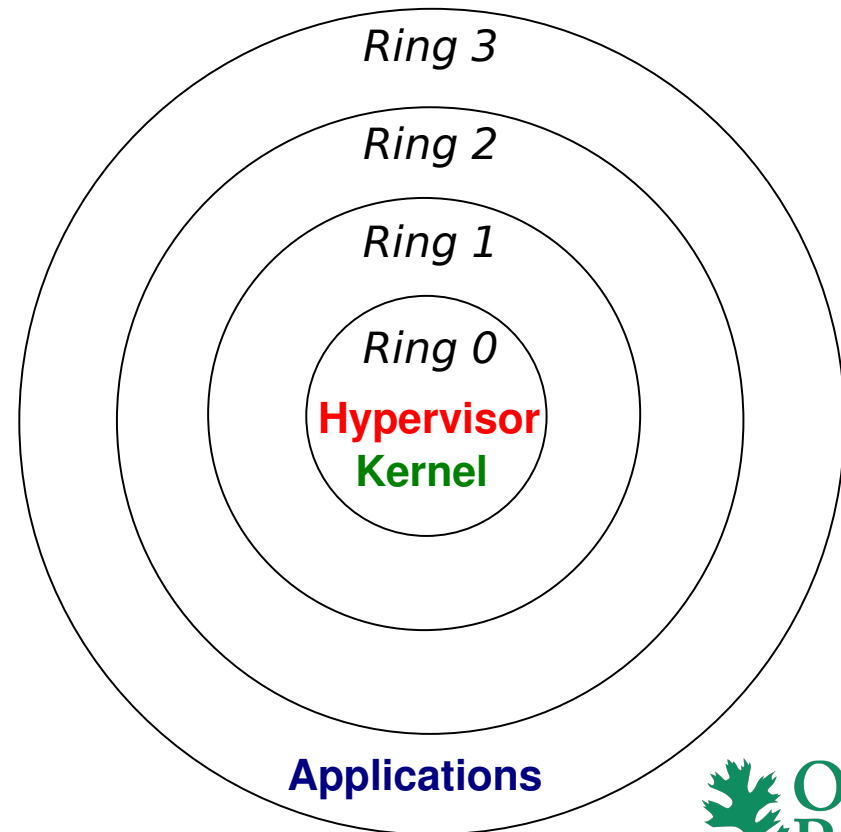
x86 Architecture - "Modified" Execution Rings

x86 Execution Rings for Hardware Protection

- *Ring 0* – hypervisor runs in this ring
- *Ring 1* – kernels run in this ring
 - Defer to hypervisor to execute protected instruction(s)
 - Hypervisor “hijack” protected processor instruction
 - Para-virtualization: hypervisor calls (hypercalls)
 - Hypercalls are like syscalls & have overhead
- *Ring 3* – applications run in this ring, no modifications

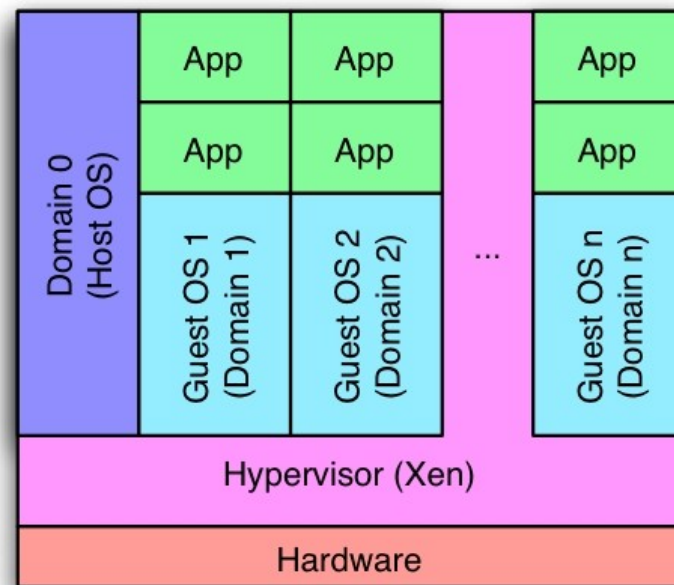
Type-I: Hardware Support

- Create a hardware “virtualized context”
- Transition from VM mode to “Hypervisor mode”
 - Save registers
 - Context switch
- Current implementations
 - Intel-VT
 - AMD-V



Type-I: Device Drivers

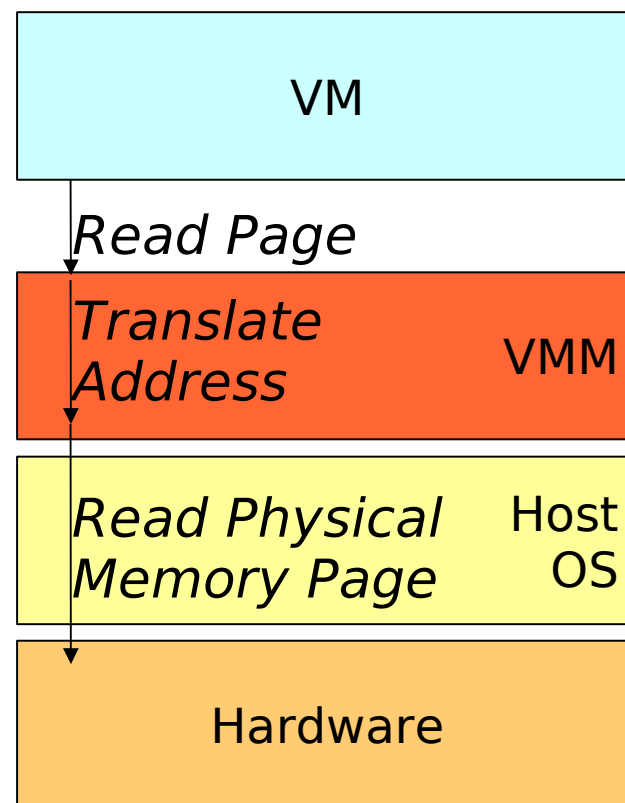
- Device drivers typically not included in the hypervisors
- Couple Hypervisor + Host OS
 - Host OS includes drivers (used by hypervisor)
 - Virtual Machines access hardware via the Host OS



Source: Barney Maccabe

Type-II: Design

- Simpler model
 - Host OS & Hypervisor are “stacked”
 - No modifications to OSES
 - Provide a BIOS simulation
- Well suited for architecture emulation
 - Ex. PPC on x86_64
- Less efficient than Type-I
 - Especially to para-virtualization



Type-I: Example

- Xen: para-virtualization (type-I)
 - Pro: good performance for computation
 - Con: overhead for I/O, modification of the Linux kernel, increasing complexity (driven by IT markets)

Type-II: Examples

- VMWare Workstation: full-virtualization (type-II)
 - Pro: mature, reasonable desktop performance
 - Con: difficult to adapt (not open source), not really suitable for HPC
- QEMU: full-virtualization (type-II)
 - Pro: open source, performance similar to VMWare, support for several architectures
 - Con: performance not suitable for HPC

Type-II: Examples (2)

- KVM: full-virtualization (type-II)
 - Pro: open source, maintained by the Linux community
 - Con: Linux as Hypervisor (size), requires Intel-VT / AMD-V
- LGUEST: full-virtualization* (type-II)
 - Pro: open source, maintained by the Linux community
 - Con: Linux as Hypervisor (size), only Linux guests**

* Only for Linux kernels

** Note: Learned at SC'07 that Ron Minnich & Plan9 work are using LGUEST to access Linux drivers – “Linux as a device driver”

Linkers, Loaders and ELFs

Background – ELF

- Executable and Linkable Format (ELF)
 - Standard for creating / interacting with object files
 - Structured into sections (segments)
 - Sections are used by linker / loaders
 - Segments are involved with execution
- ELF file types (with examples from Linux)
 - Relocatable: `e100.ko` (ethernet device driver)
 - Shared: `libc.so.6` (standard C library)
 - Executable: `vmlinux` (executable kernel)

ELF Sections

- ELF Header & Null (0'th section)
- General sections
 - .text (code), .bss (static data), etc.
- LHM section
 - .lhm.dummy_hello

ELF Header	
Magic: 7f 45 4c 46 ...	
Type: REL	
Section	NULL
Section	.text
...	
Section	.lhm.dummy_hello
...	
Section	.symtab
Section header table	
[0]	... NULL ...
...	
[7]	.lhm.dummy_hello PROGBITS ...
...	
[30]	.symtab SYMTAB ...

Background – Linker / Loader

- Linker
 - Last phase of compilation
 - Replace generic refs with actual definitions
 - e.g., `ld`
- Loader
 - Early phase of execution
 - Load into memory & relocate refs (e.g., shared libs)
 - e.g., `ld.so` or `ld-linux.so`

Customization / Adaptation

Context

- Stems from our work in virtualization for HPC
- Reasons for looking at extensibility / customization
 - Support development & performance analysis of hypervisors
 - Develop mechanism for dynamically adjusting execution environment

Motivation

- Why do dynamic customization?
 - Tailor execution environment
 - Experimentation at runtime
- Why do dynamic customization in hypervisor?
 - Perform system-level adaptation
 - Basis for future capabilities (e.g., QoS or FT)
 - Dynamic to avoid VM teardown (i.e., avoid recompile-reboot)
 - Debugging / Instrumentation
 - Add functionality as needed

Customization

- Types of customization (or adaptation)*
 - Static: compile time
 - Dynamic: run time
- Dynamic methods often used to increase functionality
 - Linux modules, i.e., add device driver
 - Live OS updates
 - Runtime instrumentation
 - Performance, debugging, steering

* Note: For now using terms “customization” & “adaptation” interchangeably

Terminology

- Customization classification [Denys:survey:dec02]
 - Initiator of adaptation (human, app, OS)
 - Time of adaptation (static, dynamic)
 - Static: design, build, install
 - Dynamic: boot, runtime
- For example, generally...
 - Xen supports: static human initiated
 - Linux supports: static human initiated & **dynamic OS initiated**

Loadable Modules

Overview

- Issue: Current Xen is very monolithic
 - Static customization via rebuild & reboot
 - To change must shutdown active VMs & save all state
 - Limits experimentation / investigation
- Idea: Extend Xen hypervisor to allow for changes at runtime, e.g., in the form of loadable modules.
- Approach: Look at details of Linux's loadable modules

Linux Modules

- Linux supports dynamic adaptation
 - Loadable Kernel Modules (LKM)
- Dynamic Loader
 - Linux 2.4: part user-space, part in-kernel
 - Linux 2.6: in-kernel loader + (small) user-space utilities
 - module-init-tools: insmod, rmmod, modprobe, etc.
- Compile additional sections into relocatable ELF “module”
 - Ex. `e100.o + e100.mod.o → e100.ko`

Steps for Linux Load Module (1)

1. Allocate user-space buffer & read in relocatable ELF object file
2. Call `init_module()`
3. Enter kernel space, `sys_init_module()`
4. Check permissions
5. Call `load_module()` to allocate space & copy data from user-space to kernel-space.
 - “Do lots of stuff” ... fixup symbols, relocate references, exception table, update instruction cache, etc.

Steps for Linux Load Module (2)

6. Add to list of modules
7. Notify system of newly loaded module
8. Initialize newly loaded module, i.e., call `mod->init()`
9. Update module state (“ready”)
10. Remove any tmp storage & final module book-keeping
11. Return to user-space, `init_module()`

Linux

User

Build module (gcc)

Load module (insmod)
↳ init_module()

Kernel

Perform sys_init_module()

Load module & fixup symbols

- copy data
- patch UNDEF symbols

Update system entries

- add to sysfs
- add to module list

If exists, run init()

Mark module as “ready”

etc...

Loadable Hypervisor Modules

- Mechanism for runtime changes to VMM
- Modeled after Linux's Loadable Kernel Modules
 - Maximize reuse to help speed development
- Maintain consistent file format
 - Relocatable ELF object with additional segment

LHM: Changes to HostOS (dom0)

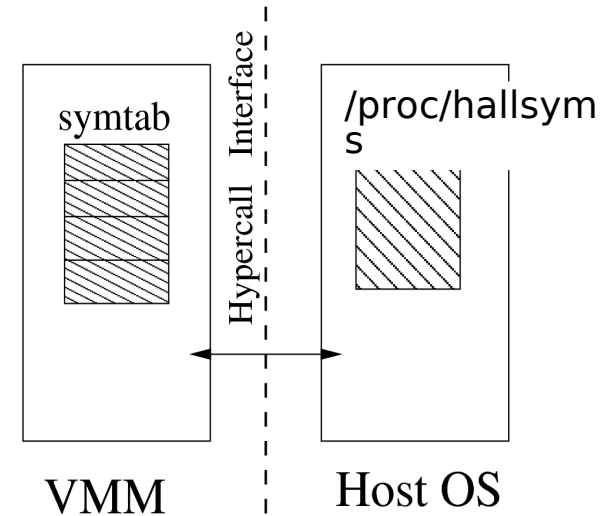
- Headers for LHM compilation
- Slightly modify module utilities (rmmod & lsmod)
- Add /proc entry to access addresses of Xen symbols
 - Export VMM symbol info to dom0
- Modify modules sub-system to detect / support LHM
 - Patch LHMs with VMM symbol info
 - Make hypercall to map LHM into VMM

LHM: Changes to VMM (xen)

- New hypercall – `HYPervisor_do_lhm_op()`
 - Map (load) LHM into VMM
 - Unload LHM from VMM
 - List loaded LHMs
- LHM “mapping” / loading
 - Allocated memory resources remain in HostOS (dom0)
 - Regions mapped into VMM & write-protected from HostOS
 - Effectively a “memory leak” from HostOS perspective (freed upon unload)

Symbols and Locations

- Linux kernel info
 - Standard feature of HostOS (dom0)
 - Provides *kernel* symbol names, types & addresses
 - Exported via `/proc/kallsyms`
- Xen hypervisor info
 - New feature of HostOS (dom0)
 - Provides *hypervisor* symbol names, types & addresses
 - Exported via `/proc/hallsyms`
 - Add new hypercall for LHM based info



Linux Example: Symbols & Addresses

- Note: External (undefined) symbols get resolved at load time.
 - Example: The kernel's `printk()` function, i.e., shared “library” routine.

```
/proc/kallsyms  
...  
c012 6100 T printk  
...  
cfad 1000 t init_module [e100]  
c012 6100 u printk [e100]  
...
```

Hypervisor Example: Symbols & Addresses

- Notice address spaces
 - Xen “ffff xxxx”
 - Linux “cfad xxxx”

```
File: /proc/hallsyms
...
ff11 3030 T printk
...
cfad 2020 T init_module [simple_lhm]
cfad 2020 T lhm_init [simple_lhm]
cfad 20e9 T cleanup_module [simple_lhm]
cfad 20e9 T lhm_exit [simple_lhm]
cfad 2136 T testme [simple_lhm]
...
ff11 3030 u printk [simple_lhm]
...
```

LHM: Before (.ko) and After (/proc)

1. Undefined symbol address resolved

```
File: simple_lhm.ko
  U printk
0000 0000 r __versions
0000 0000 D __this_module
0000 0000 D lhm_dummy
...
0000 0020 T init_module
0000 0020 T lhm_init
0000 00e9 T cleanup_module
0000 00e9 T lhm_exit
0000 0136 T testme
```

```
File: /proc/hallsyms
...
ff11 3030 T printk
...
cfad 2020 T init_module [simple_lhm]
cfad 2020 T lhm_init [simple_lhm]
cfad 20e9 T cleanup_module [simple_lhm]
cfad 20e9 T lhm_exit [simple_lhm]
cfad 2136 T testme [simple_lhm]
...
ff11 3030 u printk [simple_lhm]
...
```

2. Same offset plus absolute (actual) address

Example: Loading a LHM

- Compile LHM
 - Standard Linux build environment (kbuild)
- Load LHM
 - Standard Linux utilities (insmod)
- Enhanced Linux modules subsystem detects LHM
 - Set LHM flag in struct module
- Linux allocates space for module (in dom0)
 - Patches undefined symbols with Xen addresses
 - Ex. Xen's `printk()`
- Module is “hi-jacked”
 - Hypercall invoked to map into Xen

Host OS (dom0)

VMM (xen)

User
Kernel

Build module (gcc)

Load module (insmod)
↳ init_module()

Perform sys_init_module()

Load module & fixup symbols
- copy data
- **detect LHM**
- patch UNDEF symbols
↳ hallsyms_lookup()

Remove (skip) any Linux ties
- skip: sysfs add
- skip: module list add

Make hcall to "map" module

**Write protect memory
from Linux**

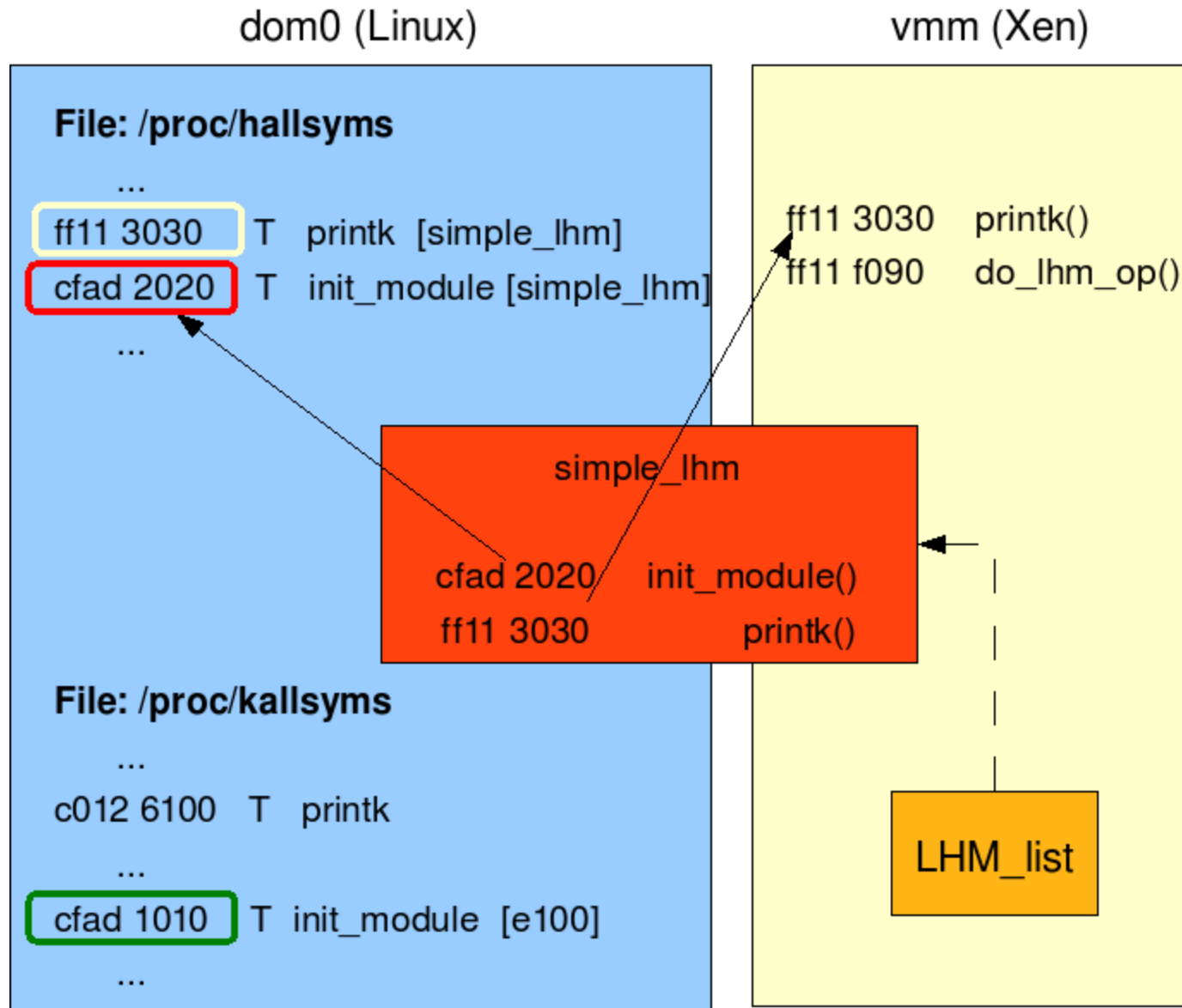
● Lookup Xen symbol info

● Perform hypercall
LHM_map_module()

- Add new LHM to list

- If exist,
call LHM's init()

Mapping of LHM



Conclusion

- Virtualization Overview
 - Types/terms, etc.
 - Focused on system-level virtualization
- Virtualization for HPC
 - Dynamic modification for HPC VMM
 - Enable new capabilities (e.g., runtime instrumentation)
- Developed new VMM mechanism
 - Based on Linux modules
 - Called Loadable Hypervisor Modules (LHM)

Future Work

- Continue LHM development
 - Stabilize and release
- Continue first LHM use case
 - Hypervisor instrumentation
 - Help in debugging & performance analysis (e.g., traces)
- Investigate other use cases
 - Adapt VMM policies at runtime (e.g., resilience?)

Questions?

- Thank you for your attention
- Thanks to AMG & SRT members
- Thanks to Ferrol Aderholdt

Supported by:

This work was supported by the U.S. Department of Energy, under Contract DE-AC05-00OR22725.

Backup / Additional Slides

- Extra slides...

Linux Module – e100.ko

- Intel PRO/100 network driver example, e100.ko
 - e100.c
 - Include module.h, vermagic.h, compiler.h
 - Declare custom sections, e.g., “.gnu.linkonce.this_module”
 - etc.
 - e100.mod.c
 - `cmd drivers/net/e100.ko := ld -m elf_i386 -m elf_i386 -r \`
`-o drivers/net/e100.ko \`
`drivers/net/e100.o drivers/net/e100.mod.o`
 - e100.o.cmd & e100.mod.o.cmd
 - Lots of stuff!!!! To compile module / stub files
 - e100.o & e100.mod.o
 - Relocatable object files

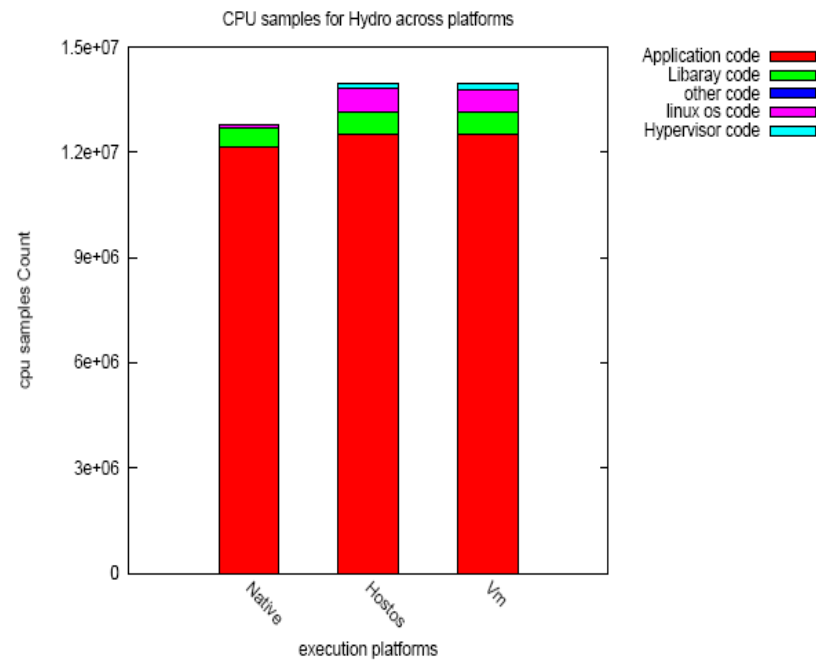
Related Work on Dynamic Modification at Systems Level

- Denys et al. [denys:survey:dec02]
 - Survey of Customization & Adaptation
- Teller & Seelam [teller:2006:osr]
 - Guidelines on dynamic OS policies
- Tamches & Miller [tamches:osdi99]
 - KernInst – dyn. instr. on commodity Solaris
 - Code splicing, etc.
- Chen et al. [chen:liveupdate:vee06]
 - live OS updates (in a VM)

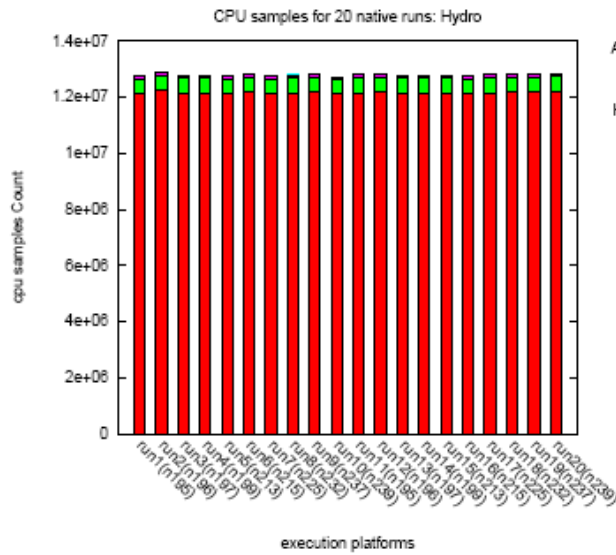
CPU time

- CPU time

- Majority of time in user code (Native & VM)
- T_{usr} roughly equiv. for Native & Virtual
- VM has ~7K more system code samples than Native

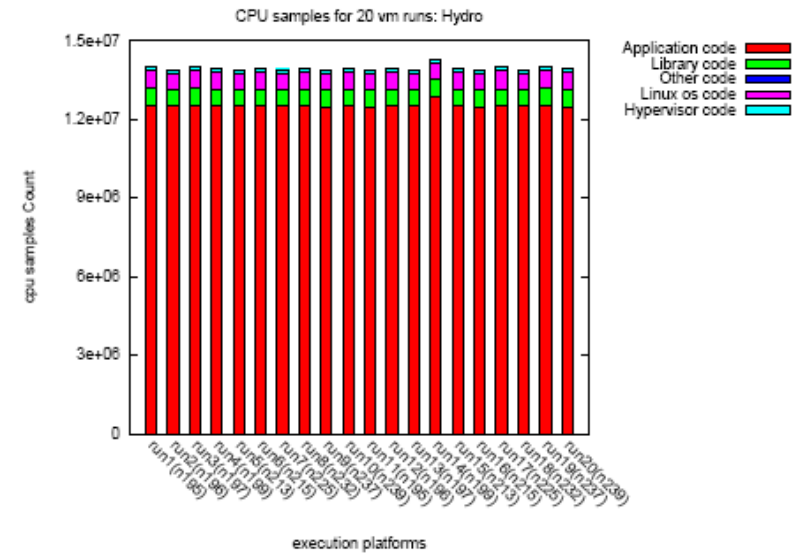


CPU: Native / HostOS / VM



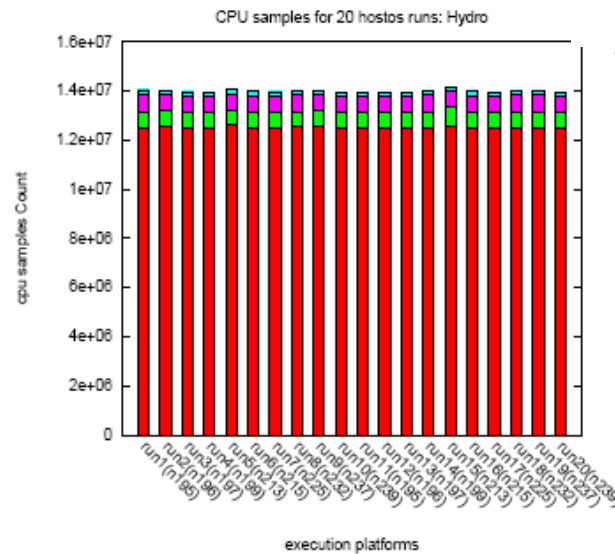
execution platforms

Native



execution platforms

VM



execution platforms

HostOS