Dynamic Adaptation using Xen:
Thoughts & Ideas on Loadable Hypervisor Modules

Thomas Naughton, Geoffroy Vallée

and Stephen L. Scott

Network and Cluster Computing
Computer Science and Mathematics Division

Outline

- Introduction
  - Terminology & Motivation
- Approach
  - Linux modules as basis for Xen modules
- Closing
  - Related work & future plans

Introduction

- Goal: Extend Xen (hypervisor) to allow for changes at runtime – *dynamic adaptation*.

- Currently Xen hypervisor very monolithic
  - Static changes via rebuild & reboot
  - Shutdown all active VMs
  - Save VM state
  - Limits experimentation/investigation

- Initial target: Xen scheduler

Terminology

- Customization & Adaptation
  - Ignoring debate of subtleties, generalize as
    - Customization is more static
    - Adaptation is more dynamic
- Customization classification [*Denys:survey:dec02*]
  - Initiator of adaptation (human, app, OS)
  - Time of adaptation (static, dynamic)

        static: design, build, install
    dynamic: boot, runtime

e.g., Xen customization: *static human initiated*

Motivation

- Why do dynamic adaptation?
  - Tailor execution environment
  - Experimentation at runtime
- Why do dynamic adaptation in hypervisor?
  - Perform system-level adaptation
  - Basis for future capabilities
    - Dynamic customization for QoS or Fault Tolerance
  - Dynamic to avoid VM teardown
    - Avoid recompile-reboot (static) customization
  - Debugging/Instrumentation
  - Add functionality as needed (microkernel ideas)

Related Work

- 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)

Approach (Plan)

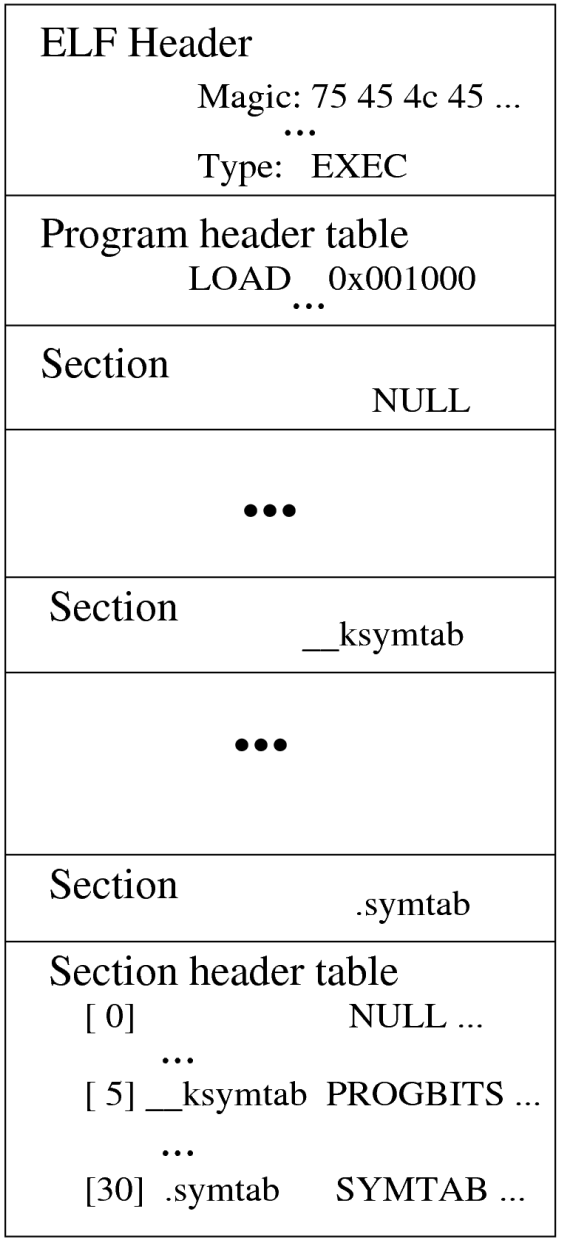
- Investigate Linux modules
  - See how/what is involved
  - Look at v2.6 “in-kernel” loader
  - Look at additional ELF segments in modules
    - Look into ELF & dynamic loader details
- Hypervisor modules
  - See if Linux approach is applicable
  - See what steps are needed
    - Access to Xen runtime symbol table, e.g., /proc
    - New hypercalls

Background - ELF

- Executable and Linkable Format (ELF)
  - Standard for creating/interacting with obj. files
  - Structured into sections/segments

- ELF file types (examples from Linux system)
  - relocatable `e100.ko` (e100 device driver)
  - shared `libc.so.6` (std C library)
  - executable `vmlinux` (executable kernel)
  - Distinctions roughly based on when symbols are resolved/relocated

Background – ELF sections/segments



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 (shared libs)
  - `ld.so` or `ld-linux.so`

Linux Modules

- Linux supports dynamic adaptation
  - Loadable Kernel Modules (LKM)
- Dynamic Loader
  - Linux-2.4 part user-space, part kernel
  - Linux-2.6 in-kernel loader + user-space insmod
    - module-init-tools
- Use standard linker to add additional sections to relocatable ELF binary (module)
  - e100.o + e100.mod.o --> e100.ko

Linux Modules – e100.ko

- Intel PRO/100 network driver example, e100.ko

  e100.c

  e100.mod.c
  - Include module.h, vermagic.h, compiler.h
  - Setup/export version magic
  - Declare custom sections, e.g., ".gnu.linkonce.this_module"
  - etc.

  .e100.ko.cmd
  - 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

Steps for Linux Install/Load Module (1)

1. allocate user-space buffer and fill with contents of relocatable ELF object (module file)

2. call init_module()

3. enter kernel space, sys_init_module()

4. check permissions & do needed locking

5. call load_module() to allocate space and copy data from user-space to kernel-space.

   Setup symbols, relocate references, exception table, update instruction cache, etc.

Steps for Linux Install/Load Module (2)

6. link module into list of available modules

7. free lock and notify system of newly loaded module

8. initialize newly loaded module

9. acquire lock and update module state (fully loaded, "ready")

10. remove any temporal storage and other final module book-keeping

11. free lock and return

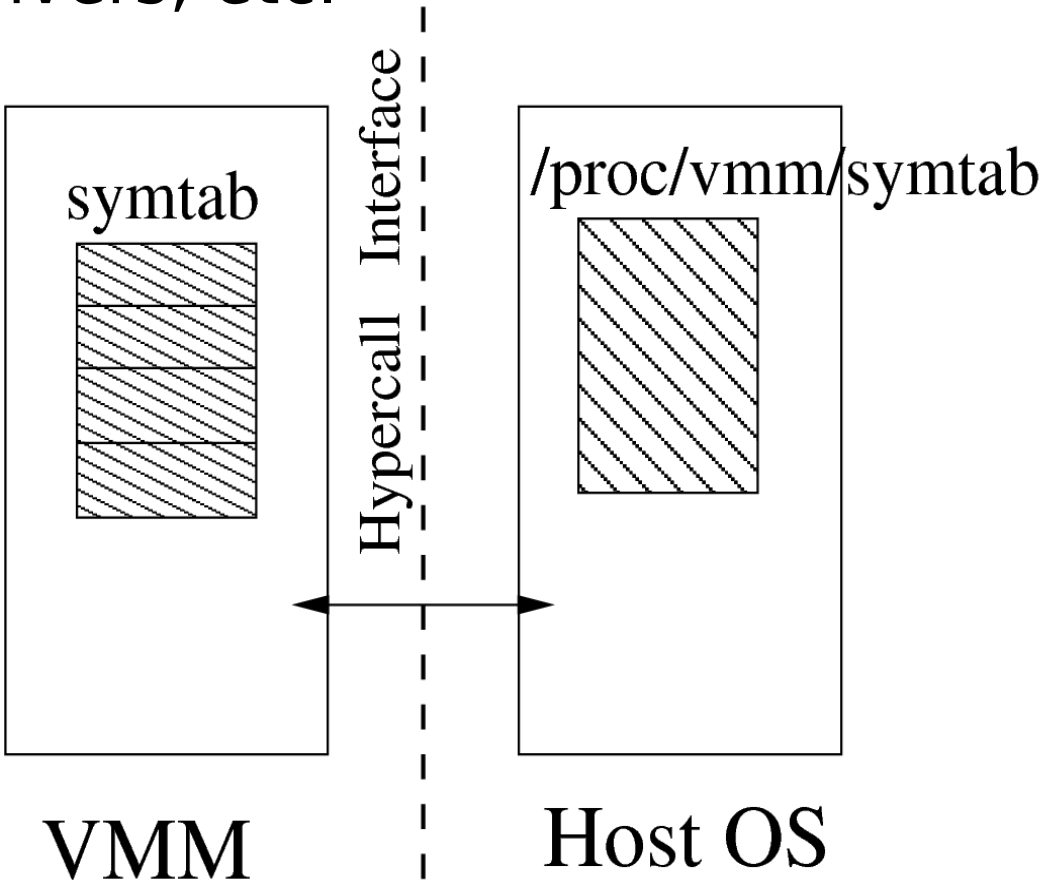
Ideas: Dynamic Hypervisor Modules (1)

- Hypervisor symbols
  - Xen symbol table
  - Exporting symbols
  - Host OS interface, e.g., /proc/vmm/symtab

- Allocation & Loading
  - Allocate space on HostOS
  - Hypercall to load into Xen space
  - in-hypervisor loader?
  - Verify (trust) modules?

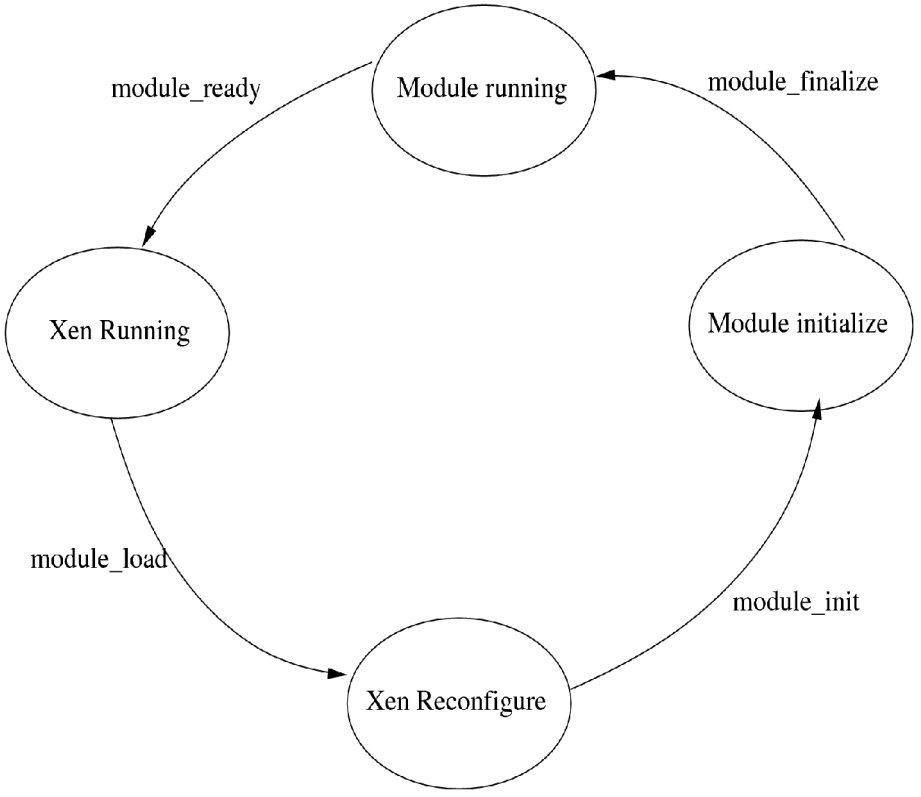
Ideas: Dynamic Hypervisor Modules (2)

- VMM and HostOS (dom0)
  - Interface to VMM
  - Device drivers, etc.



Ideas: Dynamic Hypervisor Modules (3)

- State Management
  - Module loaded, state (coming, ready, going)



Next steps…

- Xen
  - Look into exporting of symbol table
  - Dynamic instrumentation
    - Jump to resident “alternate scheduler”
      - Compiled into hypervisor to avoid dynamic ELF loading
  - ELF loader machinery

- Misc.
  - Read manuals/specs
  - Xen domain scheduler
  - Other suggestions???

Thank you

Questions / Discussion

Supported by:

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