Denali

Lightweight virtual machines

Andrew Whitaker, Marianne Shaw, Steven D. Gribble

Department of Computer Science and Engineering
University of Washington

http://denali.cs.washington.edu
{andrew, mar, gribble}@cs.washington.edu
What is Denali?

• **Denali is a research project at UW that is**
  
  – exploring the use of virtual machine monitors (think VM/370 or VMware) as isolation kernels
    
    • push untrusted service code into hosting platforms, CDNs, network measurement infrastructure, …
    
    • multiplex many services per physical machine
    
    • isolate services from each other (both security and performance)
  
  – some axis of exploration for our research
    
    • **scale**: how many services per physical node can we handle, and what gets interesting as we scale up?
    
    • **performance tunability**: how precisely and flexibly can we carve up a physical machine into virtual machines?
    
    • **recoverability/reliability**: can we exploit transparent checkpoint/migration for speculative code installation and rollback, etc.?
We have an x86 VMM running on bare hardware

- **Scale:**
  - we’ve run over 1000 VMs concurrently on single 1.5 GHz P-IV with 1 GB RAM and 90 GB disk
  - at this scale, rely on popularity distribution to allow most machines to be swapped out to disk
  - we’re doing lots of work in making this swapping smart and efficient, including detecting COW sharable pages across VMs

- **Performance:**
  - we can get 600 Mb/s out of our VMs over the network
  - compute-bound apps basically see full performance of physical machine because of VMM design
  - are just beginning to explore issues of performance isolation
    - e.g., some unexpected interactions of many TCP stacks running
But, some caveats...

- **Our VMM doesn’t support unmodified legacy OSs**
  - x86 is complex to virtualize, and we’ve shaved a few corners
    - e.g., BIOS, segmentation hardware gunk, inspecting processor status registers, …
    - but in a way that preserves security and performance isolation properties for VMs
  - there are some non-ISA architectural elements that we’ve simplified for convenience of VMs and of VMM implementation
    - boot sequence is eliminated, virtualized devices are massively simplified compared with true physical devices, …
    - implies need to change some drivers/arch-specific elements in OS.. think arch/directory in linux
  - we haven’t gotten to some things (yet)
    - virtualized page tables are the biggie: for us, so far, each VM has a single address space. If you want multiple protection domains, run multiple VMs.

- **For us, supporting legacy OS’s has been an explicit non-goal!**
  - has enabled us to explore changing virtual architecture to better support scaling, etc., and to codesign VMM with guest OS
What we’ve done instead…

- **We have put together our own real OS:**
  - ported BSD tcp/ip stack  [alpine project from UW]
  - preemptive multithreading support
  - subset of libc / POSIX API ported

- **And some applications**
  - a “supervisor” VM that can control other VMs
    - includes ability to remotely log into supervisor, and script it
  - web server
  - …
Denali and Planetlab

- **Denali could be the service virtualization layer for Planetlab**

- **Advantages**
  - research vehicle aimed at solving the very problems Planetlab faces
    - I believe it virtualizes at the right level
    - it can also easily be used to contain the volume of packets flowing out of a service if we care to do that

- **Disadvantages**
  - biggest one: doesn’t support unmodified legacy OSs, and won’t in the near future (if ever)
  - since our VMM is on the bare hardware, we only support a small # of I/O devices
    - most IDE disks, many popular 10/100 cards, some gigabit cards
  - no SMP support yet
  - we don’t have a giant customer support department
for more information…

• There are many slides appended after this with more detail. Or, visit:
  – http://denali.cs.washington.edu
    • early tech report (mostly vision) is up there
    • various conference papers are under submission or preparation, so should be appearing soon
(END OF PLANETLAB INFO, BEGIN DETAILED SLIDES)
**Denali: lightweight protection domains**

- **Goals**
  1. Watertight isolation, both in security and performance sense
     - little data or sharing between domains, hence a solution that makes sharing expensive is OK
  2. Scale up to a large number of protection domains
     - hundreds, if not thousands of services per physical machine
  3. Be able to rapidly swap between domains
     - unpopular services may only get requests once every few hours/days!
- **Our approach in Denali**
  - leverage virtual machine monitors (VMMs)
    - known to provide isolation
  - our research challenge is to make them lightweight
Infrastructural computing is on the rise

- Application functionality pushed into the network
  - services (applications) are always available, from any device

- But, barrier to deployment of new services is high
  - currently, only “big players” can afford to do it
    - cost of network infrastructure, computers, administration, …
    - stifles grassroots service innovation

- Our goal:
  *Anybody* can deploy services into network infrastructure.
Enable a wide class of new applications

- **Inject dynamic content into a CDN or cache hierarchy**
  - 20-40% of web requests are dynamic [Wolman99]

- **Allow “speculative” service deployment**
  - anybody can push a service into an ISP’s hosting platform
  - the shareware/freeware model for .NET

- **Upload filters into the network infrastructure**
  - measurement experiments on NIMI-like infrastructure
  - push virus/intrusion/DoS detection code across WAN

- **...and so on!**
Challenge 1: isolation

- **In all examples:**
  - hosting infrastructure completely distrusts uploaded services
  - services distrust each other

- **Must **iso*late** services in protection domains**
  - isolate for security
    - a service can’t hurt hosting infrastructure or other services
  - isolate performance
    - precisely “carve up” the physical resources on hosting computers
    - prevent one service from consuming unwanted share of resources
    - CPU, memory, network bandwidth, disk bandwidth, etc.
Challenge 2: issues of scale

- Lessons from the web, caches, Internet services
  - everything is driven by Zipf popularity distributions
    \[ P \text{ (access)} = k \cdot x^{-\alpha} \]
  - 50% of access to 6% sites
  - 20% of accesses to least popular 55% of sites

- Both ends of the curve matter
  - popular services: must have good performance
  - unpopular services: must be able to host many services per physical machine, and rapidly swap between them
Opportunistic goals

- The lightweight VM approach also lets us:
  - investigate transparent checkpoint, recovery
    - time-travel OS
    - “speculative” application installation
  - treat all software on a machine as cached state
    - including the OS
    - the right model for handheld devices, why not all machines?
  - enable applications of imposed architectural homogeneity
    - remote site testing and debugging
Outline

- Motivation
- Why VMMs?
- Paravirtualization and the Yakima VMM
- Some preliminary performance numbers
- Long term plans
Conventional OS view of world

- OSs provide shared abstractions and enforce protection across applications

What you're used to

- protection, abstractions, naming
- resources

sharing

operating system

hardware
The VMM approach

- **Virtualize all resources at the HW/SW interface**
  - *virtual machine monitor* (VMM) running on top of hardware

What we are doing

- Abstractions
- Protection, resources, naming

VMM

hardware
Inside a VMM

- **Virtual HW Interface**
  - VM 1
    - Virtual registers
    - Per-VM state:
      - Page tables
      - Register file
    - VNIC
    - Packet scheduler
    - Ethernet driver
  - VM 2
    - Virtual registers
    - Per-VM state:
      - Page tables
      - Register file
    - VNIC
    - Packet scheduler
    - Ethernet switch

- **VMM**
  - Swapper
  - CPU scheduler

- **HW/SW Interface**
  - Interrupt handling
  - Console
  - Timer
  - BIOS bootstrap

- **Physical Hardware (e.g., x86)**

“HAL”
VMMs provide security isolation

- **Private namespaces**
  - one VM cannot even name of another VM’s resources
  - OSs: global namespaces lead to vulnerabilities (e.g., aliases in FS)

- **Simple, static sharing policy**
  - all data is private to a VM; only sharing allowed is through network
  - OSs: complex policy across users, programs, and resources
    - nobody gets this right, ever! [Janus]

- **Protection is below abstraction**
  - VMMs have no “layer-below” vulnerabilities
    - OSs: complex implementation leads to vulnerabilities
    - e.g., shared file descriptors bypassing FS access control
VMMs enable performance isolation

- **OS: abstractions lead to “crosstalk”** [Nemesis]
  - applications compete for resources, synchronization points
  - sharing under the covers: who to charge for FS buffer cache?

- **VMMs are at the lowest (software) level of the system**
  - VMM can partition physical resources precisely
    - binding of physical to virtual resources
  - lowest level implies nothing “slips through the cracks”
    - except for HW resources we can’t affect, e.g., L1/L2 cache
  - virtual hardware devices are essentially queues
    - enables fair-queueing techniques for I/O
Challenge: scaling up a VMM

• Several things break as the # of VMs grow
  – VMM issues
    • e.g., physical resource management inside the VMM
    • rapid swapping becomes crucial
  – virtual architecture issues
    • e.g., a physical interrupt is rarely destined for the VM that is currently running
  – OS issues
    • e.g., what happens when 100 TCP stacks run simultaneously?
    • must change timers, send/receive buffer scaling, etc.

• Our lever for addressing this: para-virtualization
  – selectively modify architecture while virtualizing it
Outline

• Motivating applications
• Why VMMs?
• Paravirtualization and the Yakima VMM
• Some preliminary performance numbers
• Long term plans
Para-virtualization

- For performance, directly execute most instructions
  - VMware, VM/370: virtual arch. == physical arch.
    - run unmodified legacy OSs and applications
- Denali: VA is similar to PA, but not identical
  - added purely virtual instructions, changed some instruction semantics, added virtual instructions registers
  - simplified architectural features, changed semantics of others
- Implication: cannot run unmodified OSs
  - (but that’s not our goal!)
  - amazing opportunity: co-design virtual architecture and OS
Para-virtualization for scalability

- **Physical interrupts**
  - synchronous, imply “something just happened”
  - notification *mechanism* is conflated with interrupt *state*
    - each results in context switch, protection boundary crossing

- **Denali virtual interrupts**
  - asynchronous, imply “one or more things happened in the past”
  - single notification w/ batched interrupt state
Para-virtualization for scalability

- **Consider packet inter-arrival of an unpopular service**
  - e.g., a web session every 2 hours

- **Denali: added “idle w/ timeout” virtual instruction**
  - allows guest OS to yield CPU voluntarily

---

Denali: lightweight VMs ©2002, Steven D. Gribble
Para-virtualization for simplicity

• **There are ~17 non-virtualizable x86 instructions**
  – VMware: hairy binary rewriting/protection tricks??
  – Denali: those instructions have undefined semantics
    • if guest OS issues, can only hurt itself

• **Simplified HW devices, eliminated BIOS**
  – no boot sequence: devices are well known, pre-initialized
  – same virtual hardware on all physical hardware!
    • ability to transparently migrate VMs
  – for now, eliminated paging and CPU protection modes
    • guest OS is Exokernel-style libOS
Outline

- Motivating applications
- Why VMMs?
- Para-virtualization and the Yakima VMM
- Implementation and performance
- Long term plans
Our implementation

- **“Yakima” VMM**
  - Flux OSKIT for HAL, non-preemptive VMM kernel
  - virtual hardware support:
    - physical memory, CPU, timers, registers, NIC, disk, PIO
  - round-robin scheduling (CPU, VNICs)
  - “supervisor” VM system calls to create/destroy VMs
    - displaces complexity from VMM
    - e.g., stack for downloading VMs is in supervisor
  - swapping support, used when # VMs >> physical memory

- **“Ilwaco” guest OS**
  - subset of libC and Posix APIs, ported BSD stack
Scaling: we’re getting there…

- **Currently, our record is 1017 concurrent VMs**
  - each VM running its own OS + webserver application
  - most of them are swapped out completely at any point in time
    - a virtual hardware event triggers swap-in
    - an “active” but unpopular virtual machine requires nearly no physical resources most of the time

- **Some wrinkles happen when scaling this high**
  - ARP problems
    - ARP designed for broadcast medium, scales poorly on VMM
  - TCP timer problems
  - TCP buffer allocation issues
Context-switch overhead

- "best case": each VM touches very little memory
  - L1/L2 cache blowouts are visible
- "worst case": each VM cycles through large buffer
TCP Packet arrival overheads

- **TCP packets traverse through 4 stages:**
  1. Physical interrupt, ethernet driver processing (~6 us)
  2. Virtual NIC demultiplexing and processing (~.5 us)
  3. Copy into VM’s Rx ring buffer (~.1-.5 us)
  4. BSD TCP/IP processing (~8 us)

- **Virtualization is not the dominating factor**
  - could eliminate several redundant copies, but even that doesn’t seem to matter anymore!!
  - high cost in physical ethernet driver for 8259a PIC interaction
    - we need to move Flux OSkit over to APIC
TCP throughput / latency

Throughput (Mbps)

线性拟合: $y = 372.67x - 86.273$

Ping time (useconds)

线性拟合: $y = 372.67x - 86.273$
Web server app

- **Simple, multithreaded web server implementation**
  - all documents served out of memory
    - was built before we had disk working
  - HTTP headers precomputed and stored with content
- **Both Linux and Ilwaco can run the webserver**
  - ported Ilwaco syscall API to Linux glue library
    - didn’t need to modify the webserver code at all
  - provides “fair” comparison between virtualized and non-virtualized versions
    - not really fair: differences in stack, but let’s brush those aside
- **Used the “httpperf” client benchmarking tool from Linux**
Ilwaco web server: 1MB documents

Large document throughput

- Linux version: 551 Mb/s peak, c.f. Ilwaco’s 428
Ilwaco web server: 2KB documents

Small document latency

- Aggregate offered load (conn/s)
- Download latency (ms)

Graph showing the download latency for small documents with different aggregate offered loads and VM configurations:
- 8 VM's
- 2 VM's
- 1 VM

©2002, Steven D. Gribble
Outline

- Motivating applications
- Why VMMs?
- Para-virtualization and the Yakima VMM
- Some preliminary performance numbers
- Long term plans
On-demand loading of VMs

- **Wide-area system of demand-loaded VMs**
  - similar to caching hierarchy or CDNs
  - instead of demand-loading content, demand load an entire VM
    - same issues as cache systems, but with larger images (5-10MB instead of 5-10KB)

- **Wrinkle: what if the content-generation code relies on a large DB?**
  - either copy the DB over, or access master copy over WAN?
  - partition DB and ship slices?
    - works well for mass-customization or geographic locality
Copy-on-write tricks across VMs

• Expect that most VMs will be running same OS, if not same applications
  – VMM can copy-on-write share physical pages if they are identical across virtual machines
  – requires mechanisms for detecting identical, already resident pages when swapping in new VM

• Plan: compute checksums of code regions when swapping in
  – compare to checksums of already loaded regions
  – note this is OS, language, and object format independent!
Virtual clusters

• **Virtual clusters within a physical cluster**
  – VMMs on each cluster node cooperate with each other
  – multiple levels of resource allocation
    • resources within a node across VMs
    • nodes across virtual clusters
  – grow and shrink physical resource allocation to virtual clusters as services’ load fluctuates

• **Migration helps here as well**
  – migration can become a load balancing mechanism…
A parting thought

• **Para-virtualization blurs many lines**
  – OS / process vs. VMM / [VM:libOS]

• **some key distinctions:**
  – namespace isolation
    • no sharing of resources between VMs
  – no “layer below” issues
    • why we don’t have TCP/IP stack in VMM
  – only state in VMM is virtual device emulation state
    • simplifies migration
Questions?
1: Compare VMs with Exokernel

**Exokernel: MIT ultra-microkernel OS**
- all physical hardware names directly exposed to apps (“libOS”)
  - avoid imposing inappropriate abstractions
- resources can be shared across protection domains
  - thus, protection enforced at level of hardware
    - but below level of abstraction (disk page vs. file)
  - must map down abstraction semantics safely

**Virtual machine monitors**
- protection at same level as Exokernel (hardware)
- no high-level abstractions: expose physical names
  - but: physical names are virtualized
    - hence no sharing of resources across domains
    - avoids complexity of protection below abstraction
2: Some alternatives...

- **Simplifying policies, learning policies, etc.**
  - monitor at syscall API level
    - techniques (e.g., machine learning) to deduce OK behavior
  - appeal to simpler physical metaphors
    - WindowBox: virtual windows desktops
      - still must enforce isolation at syscall level

- **Supplement existing reference monitors**
  - Janus, TCP wrappers, software wrappers
    - Janus: hard to “compile” high level policies into filters
  - Fluke: recursive reference monitors allow policy specialization
    - but again, at OS API level
3: Compare with type-safe languages

- **Java, Modula-3: apps cannot forge references**
  - simpler to enforce access control with a reference monitor
    - example: no buffer overrun vulnerabilities!
  - but, all of these languages come with runtimes to access OS
    - security policy to protect this
    - same level-below + policy complexity flaws here

- **Virtual machine**
  - type-safety not important
    - all nameable resources inside one protection domain
    - TCB is entire virtual machine
  - abstractions on top of protected resources, not at same level
Time-multiplexed resources

• **Network bandwidth**: use fair-queuing inside the virtual Ethernet segment
  - much simpler than the normal FQ problem

• **CPU**: Use a proportional share scheduler like lottery/stride scheduling

• **Disk bandwidth is tricky because of variable disk access times**
  - use a log disk for write-intensive applications (e.g. network monitoring applications)
Some “freebies”

- **Can imagine clever virtual hardware devices**
  - copy-on-write disks, non-persistent disks
    - safely share read-only data across VMs
  - append-only log disks
    - LFS without the cleaner

- **Checkpoint / migration / recovery for free**
  - simple to capture entire machine state
    - once you can capture it, you can move it, copy it, etc.
    - all underlying hardware names are virtual
  - can even hot swap physical hardware under VMs!
“Fast boot” is a requirement

• **Issue: mechanics of swapping VMs in and out**
  – is it “APM suspend/restore”, or a “shutdown/reboot”?  
    • impacts software rejuvenation 
      – if suspend/restore, memory leaks are not cleaned up 
      – if shutdown/reboot, pay price of OS and device restart 
  – plan: suspend/restore most of the time, occasional shutdown/reboot 
    • paravirtualization helps here too: devices start in initialized state, boot sequence is minimal
Benchmarking environment

- 1.7 GHz Pentium 4, 1GB physical memory, 256KB L2
- Gigabit, switched network
  - Intel PRO/1000 card, ported linux driver version 3.0.7
  - regular (not jumbo) frames
  - default 64KB send/receive socket buffers on both ends
Para-virtualization for scalability

- Consider physical interrupt arrivals:

  virtual interrupts
  scheduled VM
  physical interrupts

- As # VMs grow:
  - virtual interrupt arrivals cluster
  - timer interrupt granularity changes [proportional to popularity]
Ilwaco web server: 1MB documents

Large document latency

Aggregate offered load (conn/s)

Download latency (ms)

8 VM's
2 VM's
1 VM

Denali: lightweight VMs ©2002, Steven D. Gribble
Ilwaco web server: 2KB documents

Small document throughput

- **Linux version:** 5490 conn/s peak, c.f. Ilwaco’s 5370
Placement of VMs inside a cluster

- goal: balanced use of physical resources to obtain maximum throughput at minimum cost ($)
  - homogenous cluster, or cluster with specialized nodes?

<table>
<thead>
<tr>
<th>service #</th>
<th>popularity</th>
<th>CPU bound</th>
<th>disk bound</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1,2</td>
<td>3,4,5</td>
<td>6 .. 50</td>
</tr>
<tr>
<td></td>
<td>{rnd}</td>
<td>{rnd}</td>
<td>51 .. 500</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>500 .. 5000</td>
</tr>
</tbody>
</table>

heterogen.

<table>
<thead>
<tr>
<th>service #</th>
<th>popularity</th>
<th>CPU bound</th>
<th>disk bound</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

homogen.