Scale and Performance in the Denali Isolation Kernel

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

Rise of the Internet Service

- **Internet services have exploded onto the scene**
  - application functionality is being pushed into the network
  - web sites, search engines, online databases, CDNs, …
- **Substantial benefits to this model**
  - services are always available, from any device
  - no software distribution: upgrades are instant
  - centralized administration by experts rather than users

But…high barrier to deployment

- **Today, service provider must manage:**
  - software infrastructure (the service itself)
  - physical infrastructure
    - computers, rack space, network bandwidth, …
- **This is a huge barrier to deployment**
  - especially for wide-area services (e.g., CDNs)
- **Challenge: can we separate these management roles?**
  - try to do for software services what has already been done for static web page hosting

A better model

- **Service pushed into third party hosting infrastructure**
- **Two main challenges**
  - **scale**: not every service will warrant an entire machine
    - Zipf's law implies half of requests to unpopular services
    - services must be ***multiplexed*** on shared physical infrastructure
  - **security**: services are untrusted
    - infrastructure cannot trust hosted services
    - services will not trust each other
    - services must be ***isolated*** in a protection domain

Isolation kernels

- **Isolation kernel: an OS-like software layer**
  - multiplexes a physical machine across a large number of isolated virtual machines
  - similar to a virtual machine monitor (e.g., VMware)
    - but, unlike VMMs, makes strategic changes to virtualized architecture for scale, performance, and simplicity

Outline

- Introduction
- **The case for isolation kernels**
  - motivating applications
  - isolation kernel design principles
- The Denali isolation kernel
  - design and implementation
- Evaluation
- Future directions and conclusions

Motivating applications

- **Dynamic content delivery in CDNs and caches**
  - avoiding the “wall” of static content
- **Building a virtual Internet service hosting center**
  - support both commercial and grassroots services
  - shareware/freeware model for .NET?
- **Wide-area Internet experimentation platform**
  - Planetlab (Culler, Peterson, et al.)
- **Commonalities:**
  - all need multiplexing and isolation
  - degree of information sharing between services is small

Design principles for isolation kernels

1. **Expose low-level resources, not high-level abstractions**
2. **Prevent direct sharing by exposing virtualized names**
3. **Zipf’s law implies a need to scale**
4. **Modify the virtual architecture for scale, performance, simplicity**

Expose low-level resources

- **OSs are ineffective at containing insecure code**
  - much of this is because of OSs' high-level abstractions
  - e.g., file system, network stack
- **Two main problems with high level abstractions**
  - layer-below attacks
    - defeat access control by requesting resource below layer of enforcement
  - significant complexity and wide API
    - no economy of mechanism: complexity hurts security
- **Instead, expose resources at level of HW/SW interface**
  - disk blocks, memory pages, NIC, …

Prevent direct sharing

- **OSs expose global namespaces to facilitate sharing**
  - but, sharing must be controlled and protected for security
  - challenge of specifying appropriate access control policy
- **Our target applications are mostly independent**
  - little direct sharing is needed!
  - expose private, virtual namespaces for low-level resources
  - the only sharing possible is share-by-copy through the network
    - trading away cheap sharing for stronger isolation

Zipf's law and the need for scale

- **Our target applications require high scalability**
  - hundreds, or thousands of services per physical machine
- **But worse, we expect Zipfian popularity distributions**
  - a few services are highly popular (easy case)
  - large fraction of requests to unpopular services
- **Need to swap services in and out of memory**
  - goals:
    - minimize in-memory footprint
    - minimize swapping overhead

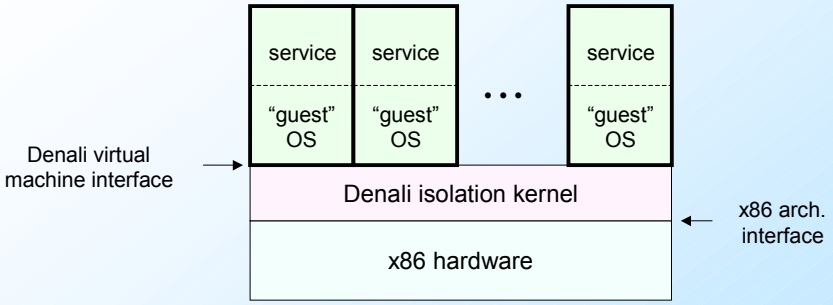
Strategically modify virtual architecture

- **So far, what we've been describing sounds like a VMM**
  - but: VMMs (VMware, VM/370) strive to support legacy systems
  - to run unmodified OSs, applications, virtualized architecture must be indistinguishable from underlying physical architecture
- **Huge burden!**
  - complexity: non-virtualizable aspects of architecture
  - performance: "chatty" device interfaces
  - scale: physical architecture not designed for scale
    - interrupt model as an example
- **Must modify virtual architecture to overcome**
  - but, giving up legacy support is a big step
    - influence design of future architectures, OSs
    - can get some of it back with OS API emulation

Outline

- Introduction
- The case for isolation kernels
  - motivating applications
  - isolation kernel design principles
- **The Denali isolation kernel**
  - design and implementation
- Evaluation
- Future directions and conclusions

The Denali isolation kernel



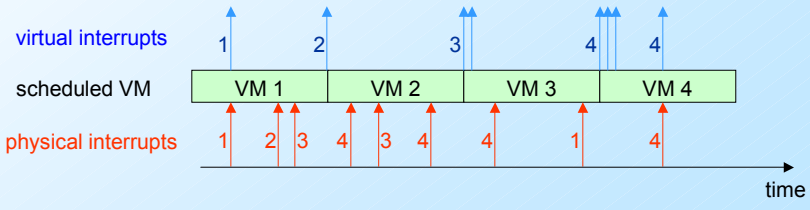
- **Two topics to cover:**
  - the Denali virtual machine interface
  - the implementation of the Denali isolation kernel

ISA

- **Denali supports a subset of the x86 ISA**
  - all virtualizable instructions are supported (direct execution)
  - non-virtualizable instructions have undefined semantics
- **Some purely virtual instructions**
  - **idle-with-timeout**: allows VM to halt its virtual CPU until a new virtual interrupt is raised, or until a timeout expires
  - **self-terminate**: allows VM to kill itself
- **New virtual registers**
  - exposes system information (CPU speed, memory size, time)
  - also used as cheap communications channel between isolation kernel and a VM
    - e.g., interrupt enabled flag

Interrupt model

- **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



Virtual architecture simplifications

- **In Denali, each VM has its own flat address space**
  - but, virtual memory is not virtualized
    - if a service needs multiple protection domains, it should be designed as multiple virtual machines
- **Virtual devices are vastly simplified**
  - virtual ethernet packet send/receive is 1 PIO
    - compared with >10 for many real NICs
  - virtual device is independent of underlying physical device
    - enhances portability
  - on boot, all devices in known, pre-initialized state
- **Other simplifications:**
  - no BIOS, no segmentation hardware

Denali implementation

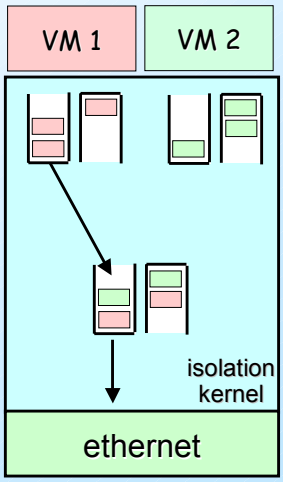
- **Denali isolation kernel runs directly on x86 hardware**
  - core of kernel (multiprogramming, paging, virtual device emulation) was built from scratch
  - used Flux OSKit for device drivers and other hardware support
- **Isolation kernel serves two roles**
  - virtualization: exposes the Denali virtual interface
  - resource management: multiplexes physical resources across virtual machines
    - we maintained a strict separation between virtualization mechanism and resource management policy

CPU and memory virtualization

- **Fairly standard mechanisms**
  - per-VM kernel thread stack, timer driven context switching, paging regions striped across disks
- **Two policies for CPU multiplexing**
  - gatekeeper: enforces admission control, by selecting a subset of active machines to admit into system
  - scheduler: controls context switching among active machines
    - round-robin scheduling
- **Swapping policy**
  - WSClock page replacement: clock variant that is more fair towards machines that are reactivated after long inactivity

Virtual I/O devices

- **A virtual I/O device is basically a queuing system**
  - virtual ethernet NIC has two queues
    - incoming (Rx) packet queue
    - outgoing (Tx) packet queue
- **Isolation kernel multiplexes and demultiplexes data from queues**
  - two policy questions:
    - what is the queueing discipline?
    - how many buffers should be allocated to each queue?



Supervisor VM

- **Denali distinguishes one VM as the "supervisor"**
  - endowed with more privilege
    - start, stop other VMs
    - VM migration: push or pull VM through supervisor
  - our supervisor VM supports remote login and scripting
- **Whenever possible, complexity displaced from isolation kernel to supervisor VM**
  - for example, there is no TCP/IP stack in isolation kernel

Library OS

- **We built our own OS to run on the Denali architecture**
  - ported BSD TCP/IP stack
    - modified timers to better support idle-with-timeout instruction
  - subset of posix UNIX API
    - libc, thread support, virtual timer wheel

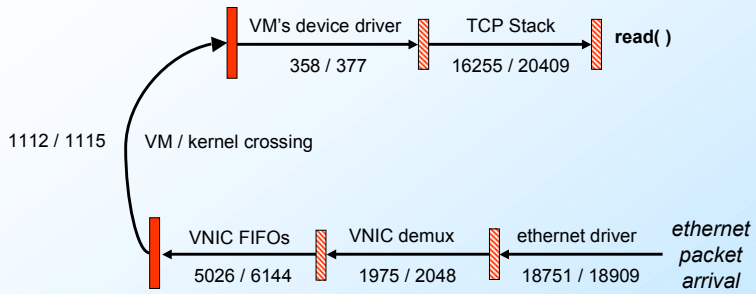
Outline

- Introduction
- The case for isolation kernels
  - motivating applications
  - isolation kernel design principles
- The Denali isolation kernel
  - design and implementation
- **Evaluation**
- Future directions and conclusions

Evaluation strategy

- **Goals**
  - demonstrate virtualization overhead is reasonable
  - evaluate impact of choices in virtual architecture
  - demonstrate scalability of system
- **All experiments done on:**
  - 1700 MHz Pentium 4
  - 1 GB RAM
  - Intel Pro/1000 gigabit card
  - three 80 GB, 7200 RPM IDE drives

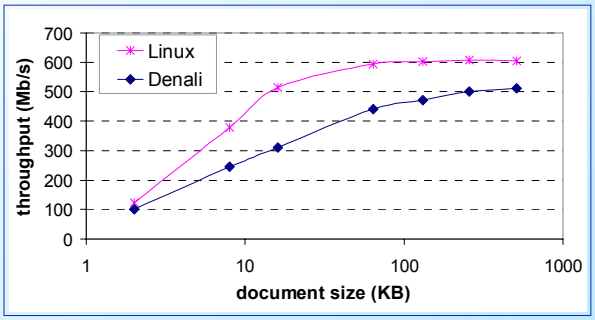
Packet dispatch latency



- **Virtualization overhead is small**
  - less than 20% of packet reception latency!

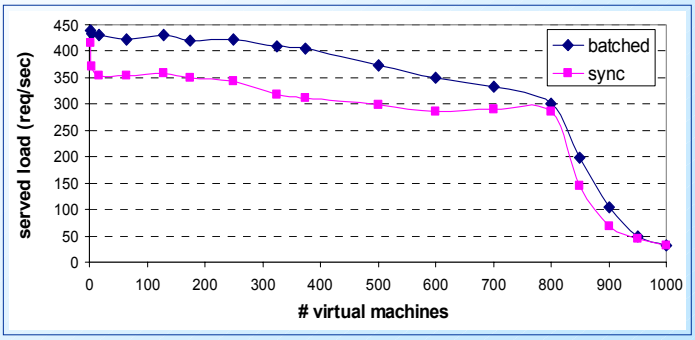
TCP, HTTP throughput

- **TCP throughput**
  - Linux-Linux: 601 Mb/s
  - Denali-Linux: 569 Mb/s (5% slower)
- **Web server throughput**



Benefits of batched interrupts

- **Compared web server running on…**
  - Denali with batched, asynchronous interrupts
  - modified Denali that immediately context switches on interrupt



Benefits of idle-with-timeout

- **64 web server VMs serving 100KB docs, compared:**
  - guest OS that exploits idle-with-timeout
    - software timer absorbs TCP retransmissions, OS idle thread invokes idle-with-timeout with smallest pending timer
  - guest OS that only calls idle when no TCP retransmissions
    - OS idle spinloops if no runnable threads, and timer pending
- **idle-with-timeout: 430 requests per second**
- **spinloop: 219 requests per second**

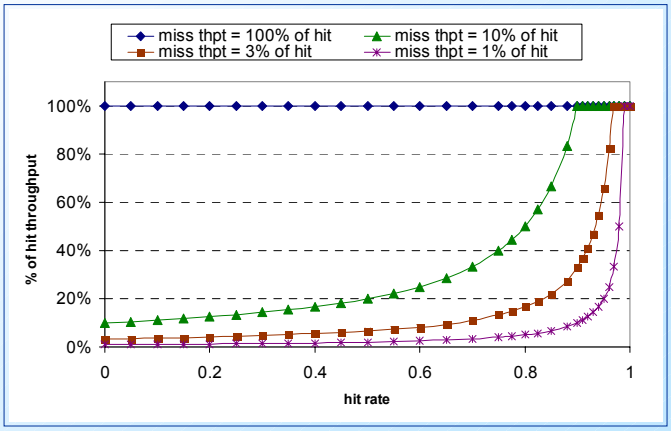
Per-VM Metadata in Isolation Kernel

- **Isolation kernel is essentially stateless (unlike an OS)**
  - scaling limit: size of metadata kept for a swapped-out VM
  - 10,000 VMs require use of about 100MB of memory

Component	Size (bytes)
thread stack	8192
register file	24
swap region metadata	20
paging metadata	40
virtual Ethernet structure	80
pending alarms	8
VM boot command line	64
other	72
Total	**8472**

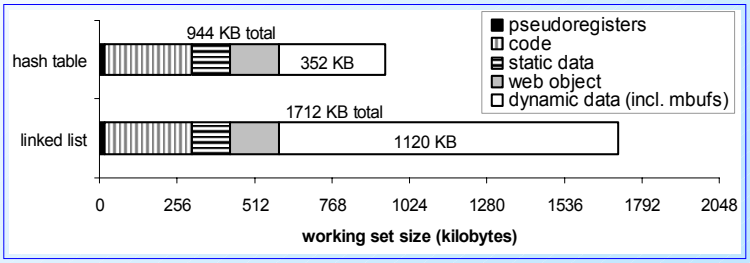
Scalability

- **Gain intuition with a simple cache model [Breslau]**
  - a “performance cliff” exists for small changes in hit rate!
  - must minimize footprint of a VM



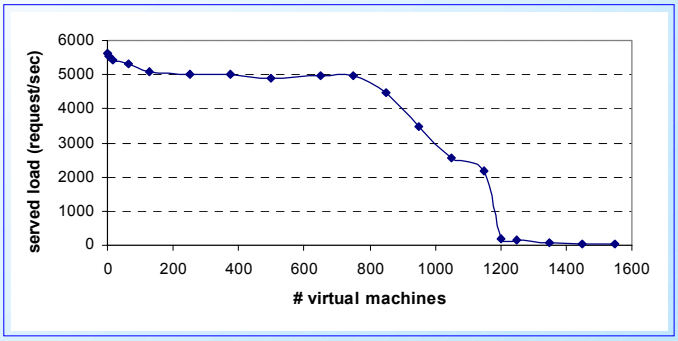
Per-VM working set size

- **Needed to redesign Mbuf pool structure in BSD stack**
  - original linked-list structure spread sequentially allocated mbufs across virtual address space over time
  - new hashed structure consolidates sequentially allocated mbufs on same virtual pages
    - factor of 2 improvement in footprint, and as a result, performance



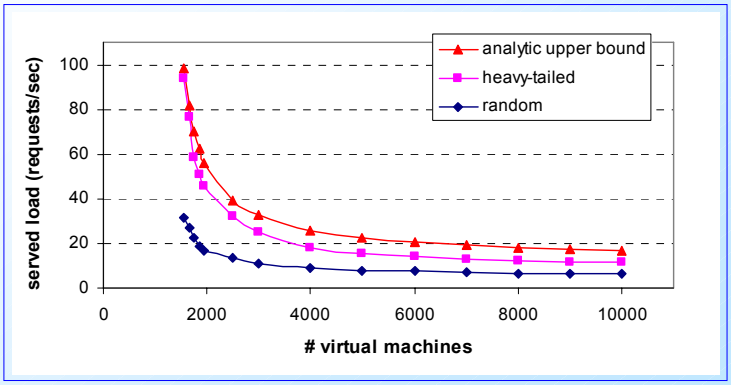
Scalability

- **Two performance regimes**
  - **all machines fit in memory**: performance degradation from context switching overhead
  - **many machines swapped to disk**: performance dominated by disk performance



Workload and scale

- For disk-bound workloads, degree of locality (value of Zipf parameter) affects performance



Outline

- Introduction
- The case for isolation kernels
  - motivating applications
  - isolation kernel design principles
- The Denali isolation kernel
  - design and implementation
- Evaluation
- Future directions and conclusions

Future Directions

- **Build and explore the motivating applications**
  - expect that an isolation kernel is only part of the solution
  - e.g., the “data issue” for dynamic content generation in CDNs
- **Virtual clusters**
  - run isolation kernel on each node in cluster
  - grow and shrink “virtual clusters” within physical cluster
- **Performance isolation**
  - fairness of resource allocation across VMs
  - independence of resource allocation policies (CPU, I/O, …)
- **Other applications**
  - untrusted device drivers in OS
  - VMs as application model for desktop OSs

Conclusions

- **Three contributions of this talk:**
  1. identification of an emerging class of applications that require multiplexing of untrusted services on shared infrastructure
  2. design principles for isolation kernels
  3. design, implementation, and evaluation of the Denali isolation kernel
- **Scale and performance is possible!**
  - required careful design of virtual architecture