OSv –

Optimizing the Operating System for Virtual Machines

Avi Kivity, Dor Laor,
Glauber Costa, Pekka Enberg,
Nadav Har'El, Don Marti, Vlad Zolotarov

Cloudius Systems

Problem statement

- Virtual Machines are useful and everywhere.
- A VM runs a **guest operating system.**
- Usually, guest OS is an existing general-purpose OS, e.g., Linux.

Can we design a better OS specifically for VMs?

Goals of OSv

OSv: a new OS designed specifically for cloud Vms.

- Run existing cloud applications (Linux executables).
- Run these faster than Linux.
- Explore new APIs for even better performance.
- Use those in a common runtime environment (e.g., JVM) to also benefit unmodified applications.

Goals of OSv (continued)

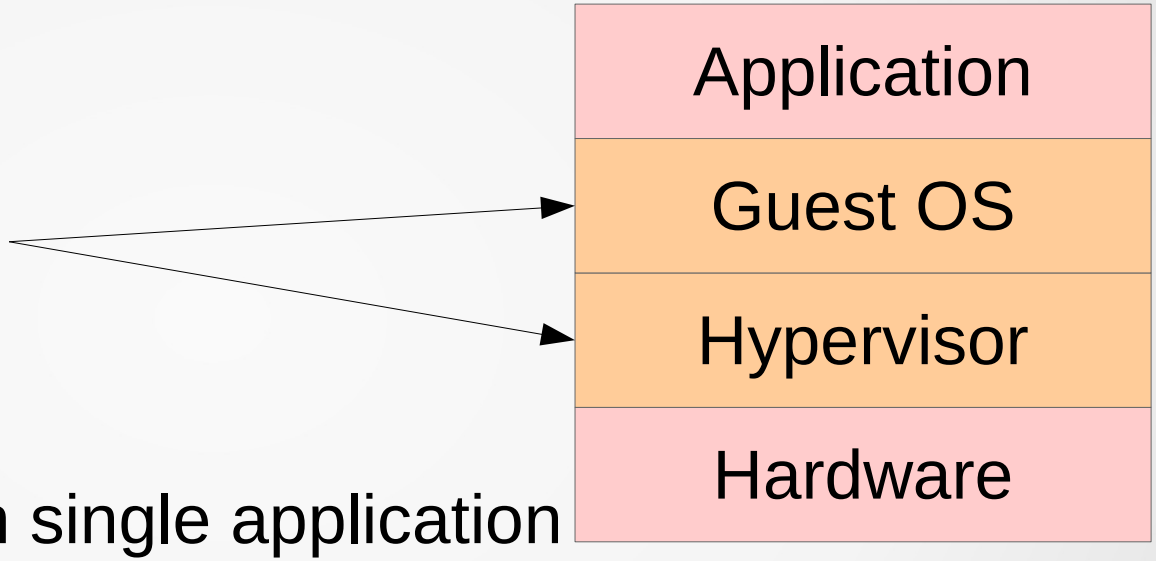
- Small image and very quick boot.
  - Starting a new VM becomes a viable alternative to reconfiguring a running one.
- Not tied to specific hypervisor or platform
  - 64-bit x86 fully working, 64-bit ARM in progress.
  - KVM, Xen, VMware, VirtualBox.
  - Amazon EC2 and Google GCE clouds.

Goals of OSv (continued)

- Be a platform for continued research on VM OSs
  - Actively developed as **open source**. http://osv.io/
  - Community encourages innovation.
  - Small code base compared to Linux.
  - Modern programming language: C++11.
  - Not limited to particular hypervisor or application programming language.
  - Fully supports SMP guests.

OSv design and implementation

- **Process isolation** is an important role of traditional OSs.
- In the cloud, both hypervisor and guest isolate applications.



- Enough for VM to run single application
  - Already common (“scale-out”).
  - Simpler code, eliminate isolation costs.

OS^{v} design and implementation

- **Single application**
  - Single process, multiple threads. Single address space.
  - No protection between user-space and kernel.
- **System calls are just function calls (Library OS)**
  - OS^{v} runs Linux shared objects by implementing an ELF dynamic linker.
  - Calls to glibc ABI are resolved to functions in the OS^{v} kernel.
  - Even “system calls”, e.g., read(), are ordinary function calls with none of the traditional system-call overheads.

OSv design and implementation

- **Linux compatibility**
  - To run existing applications, OSv implements most of the Linux/Glibc ABI.
  - Some functions like fork() and exec() are not provided, as they do not fit OSv's single-application model.

OSv design and implementation

- **No spin-locks**
  - Spin-locks are notorious for VM OSs – cause **lock holders preemption** problem.
  - Often worked around by para-virtual locks.
  - OSv avoids spin-locks **entirely**.
    - Most kernel work is done in threads, which can use a sleeping mutex.
    - Mutex implemention not using a spin-lock.
    - The scheduler uses lock-free algorithms.

OSv design and implementation

- **Network channels**
  - Network stack redesign proposed by Van Jacobson in 2006.
  - Reduce locks, lock contention and cache-line bounces.
  - Typical network stack:
    - Interrupt thread processes packets, executes TCP protocol, writes to buffer.
    - Application thread reads from this buffer.
  - Network channels:
    - Interrupt collects packets in lock-free “channels”.
    - TCP protocol executed by application thread on read().

OSv design and implementation

- The core of OSv is new code
  - Loader, Dynamic linker, Memory management, Thread scheduler, Synchronization (e.g., mutex, RCU)
  - Virtual hardware drivers:
    - PC hardware commonly emulated by hypervisors (Keyboard, VGA, IDE, HPET, etc.)
    - Paravirtual network, disk, and clock drivers (virtio, vmxnet3, pvscsi, etc.)
- Reused existing open-source code when appropriate:
  - C library headers and some functions from **Musl-libc.**
  - The ZFS filesystem from **FreeBSD**.
  - Network stack initially imported from **FreeBSD**.

Beyond the Linux API

- OS^{v} lowers the overhead of the Posix APIs.
- Some remaining overheads inherent in Posix API. E.g.,
  - read() copies data into “userspace” buffer.
  - Operations on socket lock it, as same socket can be accessed from multiple threads.
- Can we improve performance further with new APIs?

Beyond the Linux API - examples

- Zero-copy lock-less network APIs
- Direct-access to page tables
- Shrinker API: dynamic division of all of available memory.
  - JVM Balloon – automatically size JVM heap to available memory, on unmodified JVM.

Biggest obstacle to new APIs is adoption

- Can start with modifying runtime environment (JVM).
- All unmodified JVM applications would benefit.

Evaluation

- Compared OS^{v} guest to Fedora 20 guest w/o firewall.
- On KVM host.
- See full details in the paper.

Macro benchmarks

- **Memcached**. UDP. Single-vCPU guest, loaded with memaslap (90% get, 10% set)
  - OSv throughput 22% better than Linux.
- Memcached reimplemented with packet-filtering API
  - OSv throughput 290% better than baseline.
- **SPECjvm2008.** Suite of CPU/memory intensive Java workloads. Little use of OS services.
  - Can't expect much improvement. Got **0.5%**.
  - Good correctness test (diverse, checks results).

Micro benchmarks

- **Netperf** – measure network stack performance.
  - TCP single-stream thoughput: 24% improvement.
  - UDP and TCP r/r latency: 37%-47% reduction.
- **Context switch** - two threads, alternate waking each other with pthreads condition variable.
  - 3-10 times faster than in Linux.
  - As little as 328 ns when two threads on same CPU.
- **JVM Balloon** – microbenchmark where large heap and large page cache are needed, but not at the same time.
  - Osv 35% faster than Linux.

Latest unofficial results

- Experimental, non-release, code...
- Need more verification...
  - Cassandra stress test, READ, 4 vcpu, 4 GB ram
    - OSv 34% better
  - Tomcat, servlet sending fixed response, 128 concurrent HTTP connections, measure throughput. 4 vcpus, 3GB
    - OSv 41% better.

Thank you!

- Come visit us at http://osv.io/
  - Github source repository
  - Mailing list
  - Twitter, @CloudiusSystems, #Osv.
- We invite you to join the OSv open-source project!