KVM, QEMU, and Firecracker

Yiying Zhang
KVM: Linux-based Virtualization

Avi Kivity
avi@qumranet.com

Columbia University Advanced OS/Virtualization course
At a glance

- KVM – the Kernel-based Virtual Machine – is a Linux kernel module that turns Linux into a hypervisor
- Requires hardware virtualization extensions
- Supports multiple architectures: x86 (32- and 64- bit) s390 (mainframes), PowerPC, ia64 (Itanium)
- Competitive performance and feature set
- Advanced memory management
- Tightly integrated into Linux
The KVM approach

- Reuse Linux code as much as possible
- Focus on virtualization, leave other things to respective developers
- Integrate well into existing infrastructure, codebase, and mindset
- Benefit from semi-related advances in Linux
Xen

Diagram showing the structure of Xen, with layers from top to bottom:

- **Domain 0**: Includes the Xen Hypervisor and a Driver.
- **User VM**: Includes a Driver and connects to the Hypervisor through the Driver.
- **Hypervisor**: Central component managing resources and connecting the different layers.
- **Hardware**: Physical components interacting with the system.

The diagram illustrates the flow of data between these layers, emphasizing the role of the Hypervisor in mediating between the different components.
KVM Benefits

- Lightweight and efficient
- Close integration with Linux
  - Reuse Linux scheduler, memory management, etc.
- Free and well supported by the Linux community
KVM Process Model

- task
- task
guest
- task
- task
guest

kernel
/dev/kvm Device Node

- Creating a new VM
- Allocating memory to a VM
- Reading and writing virtual CPU registers
- Injecting an interrupt into a virtual CPU
- Running a virtual CPU
[recap]: Hardware-Assisted CPU Virtualization (Intel VT-x)

- Two new modes of execution (orthogonal to protection rings)
  - VMX root mode: same as x86 without VT-x
  - VMX non-root mode: runs VM, sensitive instructions cause transition to root mode, even in Ring 0

- New hardware structure: VMCS (virtual machine control structure)
  - One VMCS for one virtual processor
  - Configured by VMM to determine which sensitive instructions cause VM exit
  - Specifies guest OS state
[recap] Comparison of Pre VT-x and Post VT-x

Hardware w/o VT-x

Guest Applications

Ring 3

Guest OS

Ring 1

Hypervisor

Ring 0

Hardware w/ VT-x

Guest Applications

VMX non-root

Ring 3

Guest OS

VMX non-root

Ring 0

Hypervisor

VMX root Ring 0

Host Applications

VMX root Ring 3
[recap] VMX Mode Transition with Intel VT-x

- VM exit/entry (to/from root mode)
  - Registers and address space swapped in one atomic operation
  - Guest- and host-states saved and loaded to VMCS during transitions
- Whenever possible, sensitive instructions only affect states within the VMCS instead of always trapping (VM exit)
- VM exit
  - `vmcall` instruction
  - EPT page faults
  - Interrupts
  - Some sensitive instructions (configured in VMCS)
- VM enter
  - `vmlaunch` instruction: enter with a new VMCS
  - `vmresume` instruction: enter for the last VMCS
- Typical vm exit/enter taks ~200 cycles on modern CPU

Image source: https://www.anandtech.com/show/2480/9
KVM Execution Model

- Three modes for thread execution instead of the traditional two:
  - User mode
  - Kernel mode
  - Guest mode

- A virtual CPU is implemented using a Linux thread
- The Linux scheduler is responsible for scheduling a virtual cpu, as it is a normal thread
KVM Execution Model

- **Native Guest Execution**
- **Kernel exit handler**
- **Switch to Guest Mode**
- **ioctl()**
- **Guest**
- **Kernel**
- **Userspace**
- **Userspace exit handler**
KVM Execution Model

- Guest code executes natively
  - Apart from trap'n'emulate instructions
- Performance critical or security critical operations handled in kernel
  - Mode transitions
  - Shadow MMU
- I/O emulation and management handled in userspace
  - Qemu-derived code base
  - Other users welcome
KVM Memory Model

Figure 1: kvm Memory Map
KVM Memory Virtualization

- Supports both MMU with and without hardware-assisted two-level paging (e.g., EPT, NPT).

- When guest paging is disabled, KVM translates guest physical addresses to host physical addresses (PPN->MPN).

- When guest paging is enabled, KVM translates guest virtual addresses, to guest physical addresses, to host physical addresses (VPN->PPN->MPN).

- When the number of required translations matches the hardware, the mmu operates in direct mode; otherwise it operates in shadow mode.

[recap] Hardware-Assisted Memory Virtualization
KVM Memory Model

- Guest physical memory is just a chunk of host virtual memory, so it can be
  - Swapped
  - Shared
  - Backed by large pages
  - Backed by a disk file
  - COW'ed
- The rest of the host virtual memory is free for use by the VMM
  - Low bandwidth device emulation
  - Management code
Linux Integration

- Preemption (and voluntary sleep) hooks: preempt notifiers
- Swapping and other virtual memory management: mmu notifiers
Preempt Notifiers

- Linux may choose to suspend a vcpu's execution
- KVM runs with some guest state loaded while in kernel mode (FPU, etc.)
- Need to restore state when switching back to user mode
- Solution: Linux notifies KVM whenever it preempts a process that has guest state loaded
  - ... and when the process is scheduled back in
- Allows the best of both worlds
  - Low vmexit latency
  - Preemptibility, sleeping when paging in
Preempt notifiers

External interrupt or trap → VMM process in host kernel

Restore host state → Context switch

Restore guest state → Context switch

Context switch → Scheduler

Other process

Guest
MMU Notifiers

- Linux doesn't know about the KVM MMU
- So it can't
  - Flush shadow page table entries when it swaps out a page (or migrates it, or ...)
  - Query the pte accessed bit when determines the recency of a page
- Solution: add a notifier
  - for tlb flushes
  - for accessed/dirty bit checks
- With MMU notifiers, the KVM shadow MMU follows changes to the Linux view of the process memory map
Paravirtualization

- Yesterday's hot topic
  - Needed for decent MMU performance without two-dimensional paging
  - Intrusive

- KVM has modular paravirtualization support
  - Turn on and off as needed by hardware
  - Still needs hardware virtualization extensions

- Supported areas
  - Hypercall-based, batched mmu operations
  - Clock
Virtio

- Most devices emulated in userspace
  - With fairly low performance
- Paravirtualized I/O is the traditional way to accelerate I/O
- Virtio is a framework and set of drivers:
  - A hypervisor-independent, domain-independent, bus-independent protocol for transferring buffers
  - A binding layer for attaching virtio to a bus (e.g. pci)
  - Domain specific guest drivers (networking, storage, etc.)
  - Hypervisor specific host support
virtio: Linux’s paravirtualized I/O solution

• Front-end Driver
  • A kernel module in the guest OS
  • Accepts I/O requests from the user process
  • Transfer I/O requests to back-end driver

• Back-end Driver
  • Accepts I/O requests from front-end driver
  • Perform I/O operation via physical device

• Virtqueue
  • A memory region accessible from both guest and host OS
  • An interface implemented as vring
KVM Conclusion

- Tight integration with Linux
- The KVM module is relatively simple, with most of the functionalities already implemented in the Linux kernel
- KVM relies on hardware virtualization support (which is prevalent now)
- KVM performance is generally good
- Increasing popularity after AWS made the big move from Xen to KVM
QEMU

- Open-source Type-2 hypervisor
- Full virtualization that supports cross-architecture conversion
- Using dynamic binary translation
- Supports two modes
  - User-mode: runs Linux process in one architecture on another (host) arch
  - System emulation: runs full guest OS
QEMU Binary Translation

• Functional simulation
  • Simulate what a processor does, not how it does it
  • Supports many devices (serial, Ethernet, etc.) and many architectures

• Dynamic binary translation
  • Not an interpreter (interpreter executes one inst at a time and very slow)
  • QEMU converts code as needed and stores converted code in a translation cache
  • Code translated one block at a time

• A lot of similarities to VMware’s dynamic binary translation
Converting Code across Architectures
QEMU Tiny Code Generator (TCG)
QEMU Binary Translation

- Tiny Code Generation
- Micro-operations
- Fixed register allocation

Source: https://lugatgt.org/content/qemu_internals/downloads/slides.pdf
QEMU Dynamic Binary Translation Stage 1

- Guest Code
  - gen_intermediate_code()
    - TCG Operations
      - tcg_gen_code()
        - Host Code
          - push %ebp
          - mov %esp, %ebp
          - not %eax
          - add %eax, %edx
          - mov %edx, %eax
          - xor $0x55555555, %eax
          - pop %ebp
          - ret
QEMU Dynamic Binary Translation Stage 2

Guest Code

\[ \text{gen\_intermediate\_code()}. \]

TCG Operations

\[ \text{tcg\_gen\_code()}. \]

Host Code

\[ \ld_{i32} \text{ tmp2, env,} \text{ } 0x10 \]
\[ \text{qemu\_ld32u} \text{ tmp0, tmp2,} \text{ } 0xffffffff \]
\[ \ld_{i32} \text{ tmp4, env,} \text{ } 0x10 \]
\[ \text{movi}_{i32} \text{ tmp14,} \text{ } 0x4 \]
\[ \text{add}_{i32} \text{ tmp4, tmp4, tmp14} \]
\[ \st_{i32} \text{ tmp4, env,} \text{ } 0x10 \]
\[ \st_{i32} \text{ tmp0, env,} \text{ } 0x20 \]
\[ \text{movi}_{i32} \text{ cc\_op,} \text{ } 0x18 \]
\[ \text{exit\_tb} \text{ } 0x0 \]
QEMU Dynamic Binary Translation Stage 3

Guest Code

\text{gen\_intermediate\_code()}

TCG Operations

\text{tcg\_gen\_code()}

Host Code

\begin{verbatim}
... 
mov 0x10(%ebp),%eax
mov %eax,%ecx
mov (%ecx),%eax
mov 0x10(%ebp),%edx
add $0x4,%edx
mov %edx,0x10(%ebp)
mov %eax,0x20(%ebp)
mov $0x18,%eax
mov %eax,0x30(%ebp)
xor %eax,%eax
jmp 0xba0db428
/*This represents just the ret instruction!*/
\end{verbatim}
**QEMU Binary Translation**

- Translation block and trans cache
  - `cpu exec()` called each time around main loop. Program executes until an unchained block is encountered. Returns to `cpu exec()` through epilogue.

- Translation Block chaining

- Condition code optimization

- Async interrupt

Source: [https://lugatgt.org/content/qemu_internals/downloads/slides.pdf](https://lugatgt.org/content/qemu_internals/downloads/slides.pdf)
Memory Virtualization

- Software MMU translates virtual memory address to physical one at every memory access
- Caches the translation
- Translation blocks indexed by physical addresses
Storage Virtualization

• Application and guest kernel work similar to bare metal.
• Guest talks to QEMU via emulated hardware.

• QEMU performs I/O to an image file on behalf of the guest.
• Host kernel treats guest I/O like any userspace application.

[ source: Stefan Hajnoczi, - IBM Linux Technology Center, 2011 ]
• When the guest architecture is the same as the physical host architecture (and on a hardware-assisted virtualization platform)

• Can use KVM as the “main” hypervisor (allows native execution of guest code (in guest mode) etc.)

• Calls into QEMU when KVM cannot handle certain things (basically hardware device emulation)
QEMU+KVM

- KVM virtualizes CPU and memory (with hardware assistance)
- QEMU initiates vCPUs and assigns memory space to the VM
- QEMU emulates I/O devices
- QEMU communicates to KVM via /dev/kvm ioctl
Firecracker
Lightweight Virtualization for Serverless Applications

Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Diana-Maria Popa, and Phil Piwonka

February 2020
Why Firecracker?
EC2 m5.metal instance
384GB of RAM

Smallest Lambda Function
128MB of RAM
AWS Lambda before Firecracker

- Linux containers on VM
  - One container per function
  - One VM per customer (per machine)
- Containers: trading off security and compatibility
- VMs: difficulties of efficiently packing workloads
Requirements for Amazon

• **Isolation**: It must be safe for multiple functions to run on the same hardware

• **Overhead & Density**: Thousands of functions on a single machine

• **Performance**: Functions must perform similarly to running natively

• **Compatibility**: Arbitrary Linux binaries and libraries. No code changes or recompilation

• **Soft Allocation**: It must be possible to over-commit CPU, memory, and other resources
What about Existing Virtualization Solutions?

- QEMU/KVM: density and overhead challenges
- Linux *containers*: isolation and compatibility challenges
- LibOS approaches: compatibility challenges
- Language VM isolation: compatibility and isolation challenges
Firecracker is an open source VMM that is purpose-built for creating and managing secure, multi-tenant container and function-based services.
Firecracker Overview

- A VMM (replacing QEMU) that uses KVM to provide minimal VMs (MicroVMs)
- Supports Linux and OSv guest OS
- Rely on Linux for many functionalities
  - Saving implementation efforts
  - Fits what Amazon’s operators are familiar with
- Started with a branch of Google Chrome’s *crosvm*
  - Removed >50% code (drivers, etc.) and added more
- Written in Rust, open source
What Firecracker does not provide

- No BIOS
- Cannot boot arbitrary kernels (e.g., no Windows)
- No legacy device or PCI support
- No VM migration support
Firecracker Design Details

- Device emulation
  - Limited emulated devices, virtio for network and block devices
- One Firecracker process per MicroVM
- REST API for configure, manage, start, and stop MicroVMs
- Rate limiters for storage and network devices
- Security: apply defenses of hardware-based attacks when deploying
- Jailer: a wrapper around Firecracker to sandbox it (e.g., chroot, pid/network namespaces, seccomp with 24 whitelist syscalls etc.)

The interface b/w VM and hypervisor is the block interface

=> file system in guest OS

=> better security
Firecracker Performance

- Memory overhead of less than 5MB per container
- Boots to application code in less than 125ms
- Creation of up to 150 MicroVMs per second per host

Figure 7: Memory overhead for different VMMs depending on the configured VM size.
Firecracker + Lambda

- “Sticky” routing to few workers
- Each worker runs 100s to 1000s of slots (MicroVMs)
- If no available slot, executes “Placement” service
- Shim process in MicroVM communicates with MicroManager

Figure 2: High-level architecture of AWS Lambda event path, showing control path (light lines) and data path (heavy lines)

Figure 3: Architecture of the Lambda worker
MicroVM start latency (serial)
Discussion

• Comparing VMware ESX, Xen, KVM, what are their pros and cons? Why do you think AWS went from Xen to KVM?

• The initial goal of virtualization (allowing one type of OS/architecture to run on another type) seems to be less and less relevant nowadays. Do you agree? Can you think of some use cases where there’s still such a need?