A SLIGHTLY DIFFERENT NESTING

KVM on Hyper-V

Vitaly Kuznetsov <vkuznets@redhat.com>

FOSDEM 2018
What is nested virtualization?

In this presentation:

Windows partition
Hyper-V
Linux with KVM
Linux
Hyper-V
Windows partition

This is L2
This is L1
This is L0
Why does it matter?

- Private and public clouds (Azure) running Hyper-V
  - Partitioning ‘big’ instances for several users
  - ‘Secure containers’ (e.g. Intel Clear Containers)
  - Running virtualized workloads (OpenStack, oVirt, ...)
  - Debugging and testing
  - ...
Nesting in Hyper-V

- Introduced with Hyper-V 2016
- Main target: Hyper-V on Hyper-V
- Not enabled by default

Set-VMProcessor -VMName <VMName> -ExposeVirtualizationExtensions $true
MICRO-BENCHMARKS
Benchmark: tight CPUID loop

“Worst case for nested virtualization”

```c
#define COUNT 10000000

before = rdtsc();

for (i = 0; i < COUNT; i++)
    cpuid(0x1);

after = rdtsc();

printf("%d\n", (after - before)/COUNT);
```
Benchmark: tight CPUID loop

Results:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bare metal</td>
<td>180 cycles</td>
</tr>
<tr>
<td>L1</td>
<td>1350 cycles</td>
</tr>
<tr>
<td>L2</td>
<td>20700 cycles</td>
</tr>
</tbody>
</table>
How virtualization works (on Intel)
(simplified)

- Hypervisor prepares VMCS area (4k) representing guest state
- Hypervisor ‘runs’ the guest
- Guest runs on hardware until some ‘assistance’ is needed
- We ‘trap’ back into the hypervisor
- Hypervisor analyzes guest’s state in VMCS area and provides the required assistance
- Hypervisor modifies guest’s state in VMCS area
- Hypervisor ‘resumes’ the guest
How one may think nested virtualization works on Intel

- L0 creates VMCS for L1, runs L1
- L1 creates VMCS for L2, runs L2
- L2 traps into L1 when needed, L1 resumes L2, ...
How nested virtualization really works on Intel

(simplified)

- L1 prepares his idea of VMCS for L2
- L1 ‘runs’ L2 guest, this traps into L0
- L0 merges VMCS for L1 with L1’s idea of VMCS for L2, creates ‘real’ VMCS for L2 and ‘runs’ L2
- When ‘assistance’ is needed L2 traps into *L0*
- L0 analyzes L2 state, makes changes and resumes L1
- L1 analyzes L2 state, makes changes and resumes L2, this traps into L0
- L0 merges VMCS for L1 with L1’s idea of VMCS for L2, creates ‘real’ VMCS for L2 and ‘runs’ L2
How nested virtualization really works on Intel

(continued)

- L0 may use “Shadow VMCS” hardware feature so each VMREAD/VMWRITE instruction in L1 doesn’t trap into L0 (extremely slow otherwise)
- When L1 is done, L0 will have to copy the whole Shadow VMCS to some internal representation and re-create regular VMCS for L2 ...
- ... so this is still not very fast
Benchmark: tight CPUID loop

Solution?

- Not really, L2 VMEXITs are always going to be significantly slower compared to L1 with current Intel architecture
- ... but we can cut some corners, in particular:
  - L1 accessing and modifying L2’s VMCS
  - The need to re-create VMCS L0->L2 upon entry
- Hyper-V provides “Enlightened VMCS”
  - Store VMCS L1->L2 in a defined structure in memory, access it with normal memory reads/writes
  - “CleanFields” mask signalling to L0 which parts of VMCS really changed
  - “[PATCH 0/5] Enlightened VMCS support for KVM on Hyper-V” on the mailing list
Benchmark: tight CPUID loop

Results:

<table>
<thead>
<tr>
<th>Type</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bare metal</td>
<td>180 cycles</td>
</tr>
<tr>
<td>L1</td>
<td>1350 cycles</td>
</tr>
<tr>
<td>L2</td>
<td>8900 cycles</td>
</tr>
</tbody>
</table>
Benchmark: `clock_gettime()`

“What time is it now”

```c
#define COUNT 10000000

before = rdtsc();

for (i = 0; i < COUNT; i++)
    clock_gettime(CLOCK_REALTIME, &tp);

after = rdtsc();

printf("%d\n", (after - before)/COUNT);
```
## Benchmark: `clock_gettime()`

### Results:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bare metal</td>
<td>55 cycles</td>
</tr>
<tr>
<td>L1</td>
<td>70 cycles</td>
</tr>
<tr>
<td>L2</td>
<td>1500 (post-Meltdown/Spectre)</td>
</tr>
</tbody>
</table>
Benchmark: clock_gettime()

On L1:

```
# cat /sys/devices/system(clocksource)\nclocksource0/current_clocksource
hyperv_clocksource_tsc_page
```

On L2:

```
# cat /sys/devices/system(clocksource)\nclocksource0/current_clocksource
kvm-clock
```
Benchmark: clock_gettime()

Reason?

```c
arch/x86/kvm/x86.c:

/*
 * If the host uses TSC clock, then passthrough TSC as stable
 * to the guest.
 */
host_tsc_clocksource = kvm_get_time_and_clockread(
    &ka->master_kernel_ns,
    &ka->master_cycle_now);

ka->use_master_clock = host_tsc_clocksource && vcpus_matched
    && !ka->backwards_tsc_observed
    && !ka->boot_vcpu_runs_old_kvmclock;
```
Benchmark: clock_gettime()

Solution?

- Tell KVM Hyper-V TSC page is a good clocksource!
- But what happens when L1 is migrated and TSC frequency changes?

- Need to make L1 aware of migration
Benchmark: clock_gettime()

Solution

- ‘Reenlightenment Notifications’ feature in Hyper-V:
  - L1 receives an interrupt when it is migrated
  - TSC accesses are emulated until we update all pvclock pages for L2 guests

- See “[PATCH v3 0/7] x86/kvm/hyperv: stable clocksource for L2 guests when running nested KVM on Hyper-V”
### Benchmark: `clock_gettime()`

#### Results:

<table>
<thead>
<tr>
<th>Level</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bare metal</td>
<td>55 cycles</td>
</tr>
<tr>
<td>L1</td>
<td>70 cycles</td>
</tr>
<tr>
<td>L2</td>
<td>80 cycles</td>
</tr>
</tbody>
</table>
MACRO-BENCHMARKS
Benchmark: iperf with SR-IOV

Setup

- L1: 16 cores, 2 NUMA nodes, mlx4 VF, 4.15.0-rc8+ eVMCS/clocksource patchsets
- L2: 8 cores, 1 NUMA node, virtio-net, 4.14.11-300.fc27
Benchmark: iperf with SR-IOV

Results

![iperf total, Gbps graph](chart)

- L1
- L2
- L2 using eVMCS

Parallel threads:

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8

iperf total, Gbps
Benchmark: iperf with SR-IOV

Things to play with

- L2 -> L1 vcpu pinning
  - `<vcpupin vcpu='0' cpuset='8'/>`
- Vhost settings
  - `<driver name='vhost' txmode='iothread' ioeventfd='on' queues='2'/>`
- VF queues in L1, CPU assignment
  - MLx4 defaults are OK
- ...

Benchmark: iperf without SR-IOV

Setup

- L1: 16 cores, 2 NUMA nodes, netvsc, 4.15.0-rc8+ eVMCS/clocksource patchsets
- L2: 8 cores, 1 NUMA node, virtio-net, 4.14.11-300.fc27
Benchmark: iperf without SR-IOV

Results
Benchmark: iperf without SR-IOV

Things to play with

- L2 -> L1 vcpu pinning
- Vhost settings
- VMBus channel pinning (automatic only)

```
# lsvmbus -vv
...
VMBUS ID 17: Class_ID = {f8615163-df3e-46c5-913f-f2d2f965ed0e} - Synthetic network adapter
Device_ID = {21938293-957d-4e27-a53b-ae35f90aba2b}
Sysfs path: /sys/bus/vmbus/devices/21938293-957d-4e27-a53b-ae35f90aba2b
Rel_ID=17, target_cpu=9
Rel_ID=35, target_cpu=10
Rel_ID=36, target_cpu=3
Rel_ID=37, target_cpu=11
Rel_ID=38, target_cpu=4
Rel_ID=39, target_cpu=12
Rel_ID=40, target_cpu=5
Rel_ID=41, target_cpu=13
```

- Can be re-shuffled with “ethtool -L”
Benchmark: kernel build

Setup

- **L1**: 8 cores Intel(R) Xeon(R) CPU E5-2667 v4 @ 3.20GHz, 1 NUMA node, 30G RAM (16G tmpfs)
  - 4.14.11-300.fc27 for kernel build
  - 4.15.0-rc8+EVMCS/stable clocksource patchsets when running L2
- **L2**: 8 cores, 1 NUMA node, 30G RAM (16G tmpfs), 4.14.11-300.fc27
- **Test**: building linux kernel

```bash
# make clean && time make -j8
```
## Benchmark: kernel build

### Results:

<table>
<thead>
<tr>
<th></th>
<th>real</th>
<th>user</th>
<th>sys</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>26m42.187s</td>
<td>131m18.664s</td>
<td>21m53.760s</td>
</tr>
<tr>
<td>L2</td>
<td>26m54.887s</td>
<td>139m19.752s</td>
<td>21m31.111s</td>
</tr>
<tr>
<td>L2 (Enlightened VMCS in use)</td>
<td>27m53.110s</td>
<td>138m27.416s</td>
<td>21m3.839s</td>
</tr>
</tbody>
</table>
FURTHER IMPROVEMENTS
Nested Hyper-V features we don’t use

- Enlightened MSR bitmap
  - Natural extension of Enlightened VMCS

- Direct Virtual Flush
  - Paravirtual TLB flush for L2 guests
THANK YOU

plus.google.com/+RedHat
facebook.com/redhatinc
linkedin.com/company/red-hat
twitter.com/RedHatNews
youtube.com/user/RedHatVideos