SFO17-403: Optimizing the Design and Implementation of KVM/ARM

Christoffer Dall
“‘Efficient, isolated duplicate of the real machine’”

–Popek and Golberg

[Formal requirements for virtualizable third generation architectures ’74]
Virtualization
Hypervisor Design

Type 1 (Standalone)

VM
- App
- App
- Kernel

VM
- App
- App
- Kernel

Hypervisor

Hardware
Hypervisor Design

Type 1 (Standalone)

Type 2 (Hosted)
Hypervisor Design

Xen

Dom0
App
App
Linux

DomU
App
App
Linux

Xen
Hardware

KVM

VM
App
App
Linux

VM
App
App
Linux

KVM

Linux
Hardware
ARM Virtualization Extensions

EL0  User
EL1  Kernel
EL2  Hypervisor
ARM VE and Hypervisors

EL0
- Dom0
  - App
  - App
- DomU
  - App
  - App

EL1
- Linux
- Linux

EL2
- Xen
KVM/ARM

EL0

App
App

EL1

Linux
KVM

3. Hypercall
2. Return

EL2

KVM lowvisor

switch state

VM

App
App

Kernel

1. Hypercall
4. Return
KVM/ARM

EL0
App
App

EL1

EL2
Linux
KVM

VM
App
App

Kernel

1. Hypercall
2. Return
ARMv8.1 VHE

- Virtualization Host Extensions
- Supports running **unmodified** OSes in EL2 without using EL1
VHE: Backwards Compatible

- HCR_EL2.E2H complete enables and disables VHE
- When disabled, completely backwards compatible with ARMv8.0
- Example: Xen disables VHE
VHE: Expands Functionality of EL2

- Expanded EL2 functionality
- New registers: TTBR1_EL2, CONTEXTIDR_EL2
- New virtual EL2 timer
VHE: Support Userspace in EL0

- TGE: Trap General Exceptions
- Routes all exceptions to EL2
- VHE no longer disables EL0 stage 1 MMU
VHE: EL2&0 Translation Regime

- Same page table format as EL1
- Used in EL0 with TGE bit set
VHE: System Register Redirection

HCR_EL2.E2H == 0

mrs x0, TCR_EL1

TCR_EL1

TCR_EL2
VHE: System Register Redirection

HCR_EL2.E2H == 1

mrs x0, TCR_EL1

TCR_EL2

TCR_EL1
VHE Register Redirection

```
mrs x0, TCR_EL12
```
More VHE Register Redirection

- Some registers change bit position to be similar between EL1 and EL2
- Example: CNTHTCL_EL2 changes layout to match CNTKCTL_EL1 with extra bits
Legacy KVM/ARM without VHE

EL1

Run VM

EL2

KVM

Trap

Lowvisor

Hypervisor

Linux
KVM/ARM with VHE

Diagram:
- Linux
  - Run VM
- Hypervisor
  - KVM
  - Lowvisor
  - Function Call

EL2
Experimental Setup

*Measurements obtained using Linux in EL2. See BKK16 talk.

- AMD Seattle B0
- 64-bit ARMv8-A
- 2.0 GHz AMD A1100 CPU
- 8-way SMP
- 16 GB RAM
- 10 GB Ethernet (passthrough)
VHE Performance at First Glance

<table>
<thead>
<tr>
<th>CPU Clock Cycles</th>
<th>non-VHE</th>
<th>VHE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hypercall</td>
<td>3.181</td>
<td>3.045</td>
</tr>
</tbody>
</table>

*Measurements obtained using Linux in EL2. See BKK16 talk.
KVM/ARM Optimization #1

- Avoid saving/restoring EL1 register state
KVM/ARM Optimization #2

- Legacy KVM/ARM design enabled/disabled virtualization features on every transition
- Virtual/Physical interrupts
- Stage 2 memory translation
KVM/ARM Optimization #2

- Leave virtualization features enabled
- Host EL2 never uses stage 2 translations and always has full hardware access.
KVM/ARM Optimization #3

- Don’t context switch the timer on every exit from the VM
- Completely reworks the timer code
- 20 patches on list
KVM/ARM Optimization #4

- Reduce run loop work
- Do work in vcpu_load and vcpu_put instead
- Called when entering/exiting run-loop
- Called when preempted/scheduled
- Requires VHE
KVM/ARM Optimization #5

• Rewrite the world switch code

    kvm_arch_vcpu_ioctl_run
    {
        ...
        while (1) {
            ...
            if (has_vhe() /* static key */
                ret = kvm_vcpu_vhe_run(vcpu);
            else
                ret = kvm_call_hyp(__kvm_vcpu_run, vcpu);
            ...
        }
        ...
    }
### Microbenchmark Results

*Measurements obtained using Linux in EL2. See BKK16 talk.*

<table>
<thead>
<tr>
<th>CPU Clock Cycles</th>
<th>non-VHE</th>
<th>VHE OPT *</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hypercall</td>
<td>3.181</td>
<td>752</td>
<td>1.437</td>
</tr>
<tr>
<td>I/O Kernel</td>
<td>3.992</td>
<td>1.604</td>
<td>2.565</td>
</tr>
<tr>
<td>I/O User</td>
<td>6.665</td>
<td>7.630</td>
<td>6.732</td>
</tr>
<tr>
<td>Virtual IPI</td>
<td>14.155</td>
<td>2.526</td>
<td>3.102</td>
</tr>
</tbody>
</table>
## Application Workloads

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernbench</td>
<td>Kernel compile</td>
</tr>
<tr>
<td>Hackbench</td>
<td>Scheduler stress</td>
</tr>
<tr>
<td>Netperf</td>
<td>Network performance</td>
</tr>
<tr>
<td>Apache</td>
<td>Web server stress</td>
</tr>
<tr>
<td>Memcached</td>
<td>Key-Value store</td>
</tr>
</tbody>
</table>
Application Workloads

Normalized overhead (lower is better)

Kernbench, Hackbench, TCP_STREAM, TCP_MAERTS, TCP_RR, Apache, Memcached

non-VHE, VHE OPT*

*Measurements obtained using Linux in EL2. See BKK16 talk.
Conclusions

• Optimize and redesign KVM/ARM for VHE
• Reduce hypercall overhead by more than 75%
• Better cycle counts than x86 for key hypervisor operations
• Network benchmark overhead reduced by 50%
• Key-value store workload overhead reduced by more than 80%
Upstream Status

• Timer patches on list

• Core optimization patches coming soon