To EL2, and Beyond!
Optimizing the Design and Implementation of KVM/ARM

Christoffer Dall <cdall@kernel.org>
Shih-Wei Li <shihwei@cs.columbia.edu>
“Efficient, isolated duplicate of the real machine”

“…a statistically dominant subset of the virtual processor’s instructions be executed directly by the real processor, with no software intervention by the VMM.”

– Popek and Golberg
[Formal requirements for virtualizable third generation architectures '74]
IBM 360/91

Columbia University Computer Center machine room in February or March 1969
PDP-10

KL10 CPU and MH10 memory cabinets
Originally installed 1985 at Sikorsky Aircraft
Dual Cavium ThunderX

Gigabyte R270-T61
96 Cores
Virtualization

Non-privileged

App  App  App

Privileged

OS Kernel

Hardware

Native

Virtual Machines

Non-privileged

VM

App  App

Kernel

Hypervisor

Hardware

Privileged
Non-virtualizable architectures
ARM Hardware Virtualization Support

arm Virtualization Extensions != intel VT-x
x86 Virtualization Support

Root (Hypervisor)  Non-Root (VM)

VM Entry  VM Exit

VMCS

Ring 0 or kernel mode
Ring 1
Ring 2
Ring 3 or User Mode

Ring 0 or kernel mode
Ring 1
Ring 2
Ring 3 or User Mode
ARM Virtualization Extensions

EL0  User
EL1  Kernel
EL2  Hypervisor
EL2

- Separate CPU mode designed to run hypervisors
- Not designed to run full operating systems
- Reduced virtual memory support compared to EL1
- Limited support for interacting with userspace in EL0
ARM VE and Hypervisors

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

EL1
- Linux
- Linux

EL2
- Xen

WAX?
KVM/ARM

- KVM is integrated with Linux
- Linux is a full operating system designed to run in EL1
- KVM cannot run VMs without EL2
What if we could do this?

1. Hypercall
2. Return
ARMv8.1 VHE

- Virtualization Host Extensions
- Supports running **unmodified** OSes in EL2 without using EL1

Diagram:

- EL0: App App
- EL1
- EL2: Linux
VHE #1: Backwards Compatible

- HCR_EL2.E2H complete enables and disables VHE
- When disabled, completely backwards compatible with ARMv8.0
- Example: Xen disables VHE
VHE #2: Expands Functionality of EL2

- Expanded EL2 functionality
- Inherits all EL1 MMU features
- New virtual EL2 timer
- A corresponding EL2 system register for each EL1 system register
VHE #3: Support Userspace in EL0

- TGE: Trap General Exceptions
- Routes all exceptions to EL2
- VHE no longer disables stage 1 MMU in EL0
VHE #4: EL2&0 Translation Regime

- Same page table format as EL1
- Used in EL0 with TGE bit set
VHE #5: System Register Redirection

- Linux is written to run in EL1
- EL<x> is controlled by EL<x> system registers
- VHE runs Linux in EL2
  - **Unmodified!**
Linux is written to run in EL1

- VHE runs Linux in EL2
- **Unmodified!**
VHE: System Register Redirection

```assembly
mrs x0, ESR_EL1
```
VHE #5: System Register Redirection

mrs x0, ESR_EL1

ESR_EL1

ESR_EL2

VHE Disabled
VHE #5: System Register Redirection

VHE Enabled

```c
mrs x0, ESR_EL1
```

ESR_EL1

ESR_EL2
VHE #5: System Register Redirection

```
mrs x0, ESR_EL12
```

Diagram showing the relationship between different privilege levels (EL0, EL1, EL2) and processes (App, Kernel, Linux, KVM) in a host and virtual machine (VM) context.
VHE #5: More System Register Redirection

- Some registers change bit position to be similar between EL1 and EL2

- Example:
  - VHE: CNTKCTL_EL1 redirects to CNTHTCL_EL2
  - But they have different layouts
  - VHE: EL2 register changes layout to EL1 register (with extra bits)
Legacy KVM/ARM without VHE

EL1
- Linux
  - Run VM

EL2
- Hypervisor
  - KVM
  - Trap
  - Lowvisor
KVM/ARM with VHE
No VHE hardware

- How do we measure VHE performance?
- None available at start of this work
- Still no publicly available hardware
Modify Linux to:

1. Access EL2 registers
2. Use EL2 virtual memory system
3. Support user space applications in EL0
System Registers Accesses

-Lots of:

    #ifndef CONFIG_EL2_KERNEL
    msr tcr_el1, x0
    #else
    msr tcr_el2, x0
    #endif
EL1 VA Space (39 bits)

TTBR0_EL1

Userspace

0x0

0x7f

ffffffff

TTBR1_EL1

Kernel

0xffffffff80

0xffffffff

0xffffffff
EL2 VA Space (39 bits)

Where do we put the kernel and userspace?

TTBR0_EL2
EL2 Split VA Space

- Problem A: address space compression
- Problem B: Page table formats
- Problem C: requires TLB invalidation

*Only problems on non-VHE hardware!
Sharing Page Tables in EL0 and EL2

- Same page table between user and kernel
- Different page table format in EL0 and EL2

<table>
<thead>
<tr>
<th>Descriptor bit</th>
<th>EL0</th>
<th>EL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP[2]</td>
<td>R/W</td>
<td>R/W</td>
</tr>
<tr>
<td>AP[1]</td>
<td>User access</td>
<td>RES1</td>
</tr>
<tr>
<td>UXN/XN</td>
<td>UXN</td>
<td>XN</td>
</tr>
<tr>
<td>PXN</td>
<td>PXN</td>
<td>RES0</td>
</tr>
</tbody>
</table>
The AP[1] bit and Linux in EL2

- AP[1] controls if userspace can access the page
- Must be set to 0 for kernel mappings
- RES1 in EL2

<table>
<thead>
<tr>
<th>Descriptor bit</th>
<th>EL0</th>
<th>EL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP[2]</td>
<td>R/W</td>
<td>R/W</td>
</tr>
<tr>
<td>AP[1]</td>
<td>User access</td>
<td>RES1</td>
</tr>
<tr>
<td>UXN/XN</td>
<td>UXN</td>
<td>XN</td>
</tr>
<tr>
<td>PXN</td>
<td>PXN</td>
<td>RES0</td>
</tr>
</tbody>
</table>
RES1 definition

ARMvc8.0 hardware must treat non-register RES1 bits as:

“reads-as-written with no effect on the behaviour of the CPU”
UXN/XN and PXN for Linux in EL2

- PXN has no effect outside EL1
- UXN/XN means ‘execute never’ in both modes
- Cannot separate user and kernel executable

<table>
<thead>
<tr>
<th>Descriptor bit</th>
<th>EL0</th>
<th>EL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP[2]</td>
<td>R/W</td>
<td>R/W</td>
</tr>
<tr>
<td>AP[1]</td>
<td>User access</td>
<td>RES1</td>
</tr>
<tr>
<td>UXN/XN</td>
<td>UXN</td>
<td>XN</td>
</tr>
<tr>
<td>PXN</td>
<td>PXN</td>
<td>RES0</td>
</tr>
</tbody>
</table>
No ASID Support in EL2

- Address Space Identifiers (ASID)
- Avoids TLB aliasing by tagging accesses with per-context ID
- No ASID support in EL2
- Must invalidate EL2 TLB on host process context switch
Routing Exceptions to EL2

Linux in EL1

- EL0
  - User
  - Kernel
  - Exceptions from kernel
  - Exceptions from userspace

Linux in EL2

- EL0
  - User
  - Kernel
  - Exceptions from kernel

- EL1
  - Exceptions from userspace

- EL2
Routing Exceptions to EL2

- HCR_EL2.TGE traps general exceptions to EL2
- Does NOT work, because TGE without VHE disables MMU in userspace
Routing Exceptions to EL2

- Forward exceptions with software using a small shim
Linux in EL2 on non-VHE hardware

**The bad (and the ugly)**
- Less secure than Linux in EL1
- Relies on strictly correct implementation of RES1 page table bits
- Potentially worse performance for host workloads

**The Good**
- Good prototyping tool!
- Closely emulates performance of VHE for running VMs
Experimental Setup

*Measurements obtained using Linux in EL2.

- **AMD Seattle B0 ARM Server**
  - 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.*
The KVM Run Loop

```
while (1) {
    prepare();
    run_vcpu();
    handle_exit();
}
```
KVM/ARM Optimization

- Move logic out of the run loop and into vcpu_load and vcpu_put
- Only possible with VHE (or Linux in EL2)
ARM Generic Timers

- Also known as “Architected Timers”
- Timer hardware directly programmable by guest
- Expired timers generate physical interrupts for the hypervisor
KVM/ARM Timers

VCPU entry
  • Programs timer with guest state

VCPU is running
  • When the timer fires it causes an exit to the hypervisor

VCPU exit
  • Reads guest timer state to memory
  • Disables hardware timer
  • In software: If timer is expired, inject virtual interrupt
Optimized KVM/ARM Timers

VCPU load
• Programs timer with guest state

VCPU is running
• When the timer fires it causes an exit to the hypervisor

KVM is running
• When the time fires, the timer ISR injects virtual interrupts to the guest.

VCPU put
• Reads guest timer state to memory
• Disables hardware timer
EL1 System Registers

- Defer saving/restoring EL1 system register state to vcpu_load and vcpu_put
Virtualization Features

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

Optimized version:

- Leave virtualization features enabled
- Host EL2 never uses stage 2 translations and always has full hardware access.
Rewrite the World-Switch

- Rewrite the world switch code
- Very simple VHE function
- Complicated non-VHE function

```c
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);
        } 
    ... 
}
```
## Experimental Setup

*Measurements obtained using Linux in EL2.*

<table>
<thead>
<tr>
<th>System</th>
<th>CPU</th>
<th>Memory</th>
<th>Ethernet</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AMD Seattle B0 ARM Server</strong></td>
<td>64-bit ARMv8-A</td>
<td>16 GB</td>
<td>10 GB Ethernet (passthrough)</td>
</tr>
<tr>
<td></td>
<td>2.0 GHz AMD A1100 CPU</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8-way SMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dell r320 x86 Server</strong></td>
<td>64-bit Intel</td>
<td>16 GB</td>
<td>10 GB Ethernet (passthrough)</td>
</tr>
<tr>
<td></td>
<td>2.1 GHz Xeon E5-2450</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8-way SMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16 GB RAM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Microbenchmark Results

*Measurements obtained using Linux in EL2.*

<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)

- non-VHE
- VHE OPT*
- x86

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

- Optimize and redesign KVM/ARM for VHE
- Significant improvement in microbenchmark results
- Significant improvement in application benchmark results
- Similar (or better) performance characteristics compared to x86
- Published in USENIX ATC’17: https://www.usenix.org/system/files/conference/atc17/atc17-dall.pdf
Code

- Timer optimization patches (v4):
  https://lists.cs.columbia.edu/pipermail/kvmarm/2017-October/027836.html

- Core optimization patches:
  https://lists.cs.columbia.edu/pipermail/kvmarm/2017-October/027523.html

- Linux in EL2 (not for upstream, not supported, don’t come crying...):
  https://github.com/chazy/el2linux

- Target is v4.16

- Reviews are welcome