bhyvearm64: CPU and Memory Virtualization on Armv8.0-A

Alexandru Elisei
alexandru.elisei@gmail.com

BSDCan 2019
Ottawa, Canada
May 2019
About Me

- Bachelor’s Degree in Computer Science from University POLITEHNICA of Bucharest (Summer of 2018)
- bhyvearm64 was the subject of my Bachelor’s thesis
Outline

Project Overview

CPU Virtualization

Memory Virtualization

Future Work

Demo
Outline

Project Overview

CPU Virtualization

Memory Virtualization

Future Work

Demo
What is bhyvearm64?

- Type 2 hypervisor for FreeBSD
- Virtualization on Armv8.0-A
- Based on bhyve for x86 and Armv7
- Not yet integrated with the FreeBSD kernel

1https://arm.com/fvp
bhyvearm64

► What is bhyvearm64?
  • Type 2 hypervisor for FreeBSD
  • Virtualization on Armv8.0-A
  • Based on bhyve for x86 and Armv7
  • Not yet integrated with the FreeBSD kernel

► What can it do?
  • Run a FreeBSD virtual machine on the Foundation Platform\(^1\)
  • It can use virtio-mmio for network and block devices

\(^1\)https://arm.com/fvp
What is bhyvearm64?
• Type 2 hypervisor for FreeBSD
• Virtualization on Armv8.0-A
• Based on bhyve for x86 and Armv7
• Not yet integrated with the FreeBSD kernel

What can it do?
• Run a FreeBSD virtual machine on the Foundation Platform\(^1\)
• It can use virtio-mmio for network and block devices

What it cannot do?
• Create multiple virtual CPUs for a virtual machine
• Has not been run on real hardware

\(^1\)https://arm.com/fvp
Arm is Coming

Arm wants to enter the server market

Arm is Coming

Arm wants to enter the server market

- Amazon AWS Graviton CPUs (based on Cortex-A72)

---

Arm is Coming

Arm wants to enter the server market

- Amazon AWS Graviton CPUs (based on Cortex-A72)
- Arm Neoverse N1 CPUs (server-specific microarchitecture)

---

Arm is Coming

Arm wants to enter the server market

- Amazon AWS Graviton CPUs (based on Cortex-A72)
- Arm Neoverse N1 CPUs (server-specific microarchitecture)
- Cavium ThunderX2:
  “The results presented in this paper demonstrate that Arm-based processors are now capable of providing levels of performance competitive with state-of-the-art offerings from the incumbent vendors, while significantly improving performance per Dollar.”

---

bhyvearm64 Architecture

**User space**
- bhyveload
- bhyve
- bhyvectl
- libvmmapi

**Kernel space**
- `/dev/vmm/<vmname>`
- vmm

FreeBSD
bhyveload

bhyveload  [−b <memory−base−address >]  \
[−m <mem−size >]  \\n[−k <kernel−image >]  \\
[−l <load−address >]  \\
[−e <name=value >]  \\
<vmname>

Guest virtual hardware: devicetree
bhyve

```
bhyve  -s '0x200@0x7000 #24: virtio -blk, virtio.img' \
       -s '0x200@0x6000 #23: virtio -net, tap0' \
       -b

<vmname>
```
bhyvectl

bhyvectl  --vm=<vmname>  --destroy
Outline

Project Overview

CPU Virtualization

Memory Virtualization

Future Work

Demo
Challenges:

▶ The host must control the guest
Challenges:

- The host must control the guest

Motivation: Popek and Goldberg’s safety requirement for virtualization
Challenges:

- The host must control the guest

Motivation: Popek and Goldberg’s safety requirement for virtualization

Solution: use Exception Level 2 (EL2)
CPU execution mode for virtualization
EL2

CPU execution mode for virtualization

▶ Shared general purpose registers
EL2

CPU execution mode for virtualization

- Shared general purpose registers
- Different virtual address space
EL2

CPU execution mode for virtualization

- Shared general purpose registers
- Different virtual address space
- New system registers that control EL2
EL2

CPU execution mode for virtualization

- Shared general purpose registers
- Different virtual address space
- New system registers that control EL2
- New system registers that control EL1&EL0
EL2 Limitations in Armv8.0

- Different system register names
EL2 Limitations in Armv8.0

- Different system register names
- Virtual address space that cannot be shared with userspace
EL2 Limitations in Armv8.0

- Different system register names
- Virtual address space that cannot be shared with userspace
- Synchronous exceptions are normally routed to EL1
EL2 Limitations in Armv8.0

- Different system register names
- Virtual address space that cannot be shared with userspace
- Synchronous exceptions are normally routed to EL1
- It is optional
EL2 Limitations in Armv8.0

- Different system register names
- Virtual address space that cannot be shared with userspace
- Synchronous exceptions are normally routed to EL1
- It is optional

Conclusion: the host must run at EL1
hypervisor Architecture

FreeBSD Host

VM

EL0

Userspace

EL1

Kernel

EL2

vmm

HARDWARE
Challenges (continued)

Challenges:

▶ The host must control the guest

▶ Alternate execution between the host and the guest
Challenges (continued)

Challenges:

- The host must control the guest
- Alternate execution between the host and the guest

Solution: save CPU state
Challenges:

▶ The host must control the guest
▶ Alternate execution between the host and the guest

Solution: save CPU state

CPU state:

▶ General purpose registers
▶ EL1 system registers
▶ EL2 system registers that control execution at EL1&EL0
World Switch

World switches are executed:
- From guest to host
- From host to guest
World Switch

World switches are executed:
- From guest to host
- From host to guest

Save CPU state:
- EL2 stack for host CPU state
- One `struct hypctx` for each virtual CPU
Challenges (continued)

Challenges:

- The host must control the guest
- Alternate execution between the host and the guest
- The host must execute privileged instructions on behalf of the guest
Challenges:

- The host must control the guest
- Alternate execution between the host and the guest
- The host must execute privileged instructions on behalf of the guest

Motivation: Popek and Goldberg’s fidelity requirement for virtualization
Challenges (continued)

Challenges:

- The host must control the guest
- Alternate execution between the host and the guest
- The host must execute privileged instructions on behalf of the guest

Motivation: Popek and Goldberg’s fidelity requirement for virtualization

Solution: host access to guest state
Running in EL2

Goals:

▶ Execute code
▶ Share memory with host running in EL1
Running in EL2

Goals:
- Execute code
- Share memory with host running in EL1

Running in EL2:
- Create function wrapper for the HVC assembly instruction
Running in EL2

Goals:
- Execute code
- Share memory with host running in EL1

Running in EL2:
- Create function wrapper for the HVC assembly instruction
- Replace EL2 exception vector table
Running in EL2

Goals:

▶ Execute code
▶ Share memory with host running in EL1

Running in EL2:

▶ Create function wrapper for the HVC assembly instruction
▶ Replace EL2 exception vector table
▶ Map the code in the EL2 address space
Running in EL2

Goals:

▶ Execute code
▶ Share memory with host running in EL1

Running in EL2:

▶ Create function wrapper for the HVC assembly instruction
▶ Replace EL2 exception vector table
▶ Map the code in the EL2 address space
▶ Convention: first argument is the function to execute
EL0, EL1 and EL2 address spaces
Sharing Memory

- **Kernel VA Range**: 0x00000000 00000000 to 0x0000FFFF FFFFFFFF
- **Identity mapping**: 0x00007FFF FFFFFFFF
- **TTBR0_EL2**: 0x00000000 00000000

![Diagram showing memory mapping and ranges](image-url)
Outline

Project Overview

CPU Virtualization

Memory Virtualization

Future Work

Demo
Challenges:

▶ The guest assumes it has access to the entire physical memory
▶ The host must control the entire physical memory
Challenges:

▶ The guest assumes it has access to the entire physical memory
▶ The host must control the entire physical memory

Solution: stage 2 translation
Translation Regimes

Host

Stage 1 translation

PA

Guest

Stage 1 translation

IPA

Hypervisor

Stage 2 translation

PA

FreeBSD
Stage 1 Translation with 4K Pages

- **VA (64 bits)**
  - Subrange selector (16 bits)
  - T0 index
  - T1 index
  - T2 index
  - T3 index
  - Page offset (12 bits)

- **Key:**
  - TD - Table Descriptor
  - PD - Page Descriptor

- **PA (variable)**
  - Page offset

Diagram shows the translation process from VA to PA through levels 0 to 3, with TD and PD boxes indicating the table and page descriptors at each level.
Stage 2 Translation with 4K Pages

IPA

Level 1 table

TD

T1 index

Level 2 table

TD

T2 index

PD

Page offset (12 bits)

T3 index

Level 3 table

PA

Page offset

Key:
TD - Table Descriptor
PD - Page Descriptor
Stage 1 vs Stage 2

Differences between stage 1 and stage 2 translation:

- Stage 2 has 3 tables (vs 4)
Stage 1 vs Stage 2

Differences between stage 1 and stage 2 translation:
- Stage 2 has 3 tables (vs 4)
- Stage 2 starts at the level 1 table (vs level 0)
Stage 1 vs Stage 2

Differences between stage 1 an stage 2 translation:

- Stage 2 has 3 tables (vs 4)
- Stage 2 starts at the level 1 table (vs level 0)
- Stage 2 has a variable level 1 table size
4K Page Descriptors

Stage 1 page descriptor

Stage 2 page descriptor

Legend

Identical

Different
Stage 1 vs Stage 2 (continued)

Differences between stage 1 and stage 2 translation:

- Stage 2 has 3 tables (vs 4)
- Stage 2 starts at the level 1 table (vs level 0)
- Stage 2 has a variable level 1 table size
- Different format for the page table entries
Stage 1 vs Stage 2 (continued)

Differences between stage 1 and stage 2 translation:

- Stage 2 has 3 tables (vs 4)
- Stage 2 starts at the level 1 table (vs level 0)
- Stage 2 has a variable level 1 table size
- Different format for the page table entries

Solution: implement a new translation table format
A new translation table type:

```c
enum pmap_type {
    PT_STAGE1,
    PT_STAGE2,
    PT_INVALID,
};
struct pmap {
    ...
    enum pmap_type pm_type;
};
```

Existing functions take into account the table type:

```c
int pmap_enter(pmap_t pmap, ...) {
    ...
    if (pmap->pm_type == PT_STAGE1) {
        /* Create stage 1 page */
    } else {
        /* Create stage 2 page */
    }
    /* Create stage 2 page */
}
...
VM memory

```
bhyveload -k kernel.img -b 0x1000 -m 128MB \ example_vm
```
VM memory

```
bhyveload -k kernel.img -b 0x1000 -m 128MB \ example_vm
```
bvmconsole

bhyve -b example_vm
bvmconsole

bhyve -b example_vm

Stage 1 translation → IPA → Stage 2 translation

VA → IPA

BVM_CONS_PORT → Data abort

PA
Outline

Project Overview

CPU Virtualization

Memory Virtualization

Future Work

Demo
Future Work

Merge bhyvearm64

- Split bhyve for x86 into machine independent (MI) and dependent (MD) code
- User space improvements
- Hypervisor improvements
- Better validation
MI/MD Split

Promote code reuse
MI/MD Split

Promote code reuse

- Split libvmmapi (D17874)
- Split bhyve, bhyveload and bhyvectl
- Split the vmm kernel module
User Space Improvements

- Use virtio-pci
User Space Improvements

- Use virtio-pci
- Emulate the NS16550A UART
User Space Improvements

- Use virtio-pci
- Emulate the NS16550A UART
- Emulate USB
Hypervisor Improvements

- Implement SMP
Hypervisor Improvements

- Implement SMP
- Implement Virtual Host Extensions (VHE) (Armv8.1)
Hypervisor Improvements

- Implement SMP
- Implement Virtual Host Extensions (VHE) (Armv8.1)

**Figure:** Without VHE
Hypervisor Improvements

- Implement SMP
- Implement Virtual Host Extensions (VHE) (Armv8.1)

Figure: Without VHE

Figure: With VHE
Better Validation

- **KVM-unit-tests**
  - Patches for using NS16550A UART and PSCI
  - Same boot protocol as the Linux kernel

---

3 https://www.linux-kvm.org/page/KVM-unit-tests
Better Validation

- KVM-unit-tests\(^3\)
  - Patches for using NS16550A UART and PSCI
  - Same boot protocol as the Linux kernel
- Boot Linux as a guest

\(^3\)https://www.linux-kvm.org/page/KVM-unit-tests
Outline

Project Overview

CPU Virtualization

Memory Virtualization

Future Work

Demo
Acknowledgements

- Mihai Carabaş (technical advisor)
- Mihai Darius (virtio-mmio)
- FreeBSD Foundation (financial support)
Questions?
Questions?

Thank you!

▶ Project repository:
https://github.com/FreeBSD-UPB/freebsd/tree/projects/bhyvearm64
▶ Scripts and tutorial:
https://github.com/FreeBSD-UPB/bhyvearm64-utils