Porting bhyve on ARM

Mihai Carabas, Peter Grehan
{mihai,grehan}@freebsd.org

bhyvecon Tokyo 2016 - The BSD Hypervisor Conference
Tokyo University of Science
Tokyo, Japan
March 11th, 2016

About me

- University POLITEHNICA of Bucharest
  - PhD Student: virtualization on embedded devices
  - Teaching Assistant: operating systems, systems architecture, networks

FreeBSD

About me

- University POLITEHNICA of Bucharest
  - PhD Student: virtualization on embedded devices
  - Teaching Assistant: operating systems, systems architecture, networks
- BSD world
  - DragonFly BSD: SMT aware scheduler - 2012, Intel EPT for vkernels - 2013
  - FreeBSD - bhyve: instruction caching - 2014, porting bhyve on ARM - 2015 (and present)

About me

- University POLITEHNICA of Bucharest
  - PhD Student: virtualization on embedded devices
  - Teaching Assistant: operating systems, systems architecture, networks
- BSD world
  - DragonFly BSD: SMT aware scheduler - 2012, Intel EPT for vkernels - 2013
  - FreeBSD - bhyve: instruction caching - 2014, porting bhyve on ARM - 2015 (and present)
- Promoting bhyve through some diploma and master projects related to bhyve (e.g. ATA emulation)
- Coordinating these diploma and master projects

Hardware Assisted Virtualization

- a new CPU privilege level
  - on Intel/AMD: extends the current kernel mode (root/non-root)
  - on ARM: a brand new level called Hyp-mode

FreeBSD

Hardware Assisted Virtualization

- a new CPU privilege level
  - on Intel/AMD: extends the current kernel mode (root/non-root)
  - on ARM: a brand new level called Hyp-mode
- Type-2 hypervisor on ARM is more difficult to achieve
  - have to rewrite significant parts of the base OS to use the new registers
  - even then you can't run userspace apps directly over it

FreeBSD

Type-2 hypervisor on ARM

- We need to leverage the FreeBSD management mechanisms
- Don't want to write a full hypervisor from scratch

Type-2 hypervisor on ARM

- We need to leverage the FreeBSD management mechanisms
- Don't want to write a full hypervisor from scratch
- Insert only a small code into Hyp-mode
  - bridge between the Host-OS and the hardware
  - it's called when doing hypervisor operations

FreeBSD

Type-2 hypervisor on ARM

- We need to leverage the FreeBSD management mechanisms
- Don't want to write a full hypervisor from scratch
- Insert only a small code into Hyp-mode
  - bridge between the Host-OS and the hardware
  - it's called when doing hypervisor operations
- Other type-2 implementation - KVM
  - VirtualOpenSystems did the same thing

FreeBSD

Current status work

- running with bhyve a FreeBSD virtual machine
- output through a paravirtualized serial console
- it's getting to starting the init process
- but the VM is flooded with spurious interrupts

FreeBSD

Steps I've taken

- crafted an init code placed in `locore`
  - it jumps to a routine where it checks if the platform booted in Hyp-mode
  - install some stub exception vector for Hyp-mode
  - marks the virtualization available

FreeBSD

Steps I've taken (2)

- created a new `sys/arm/vmm`
- copied the VMM interface from `sys/amd64/vmm`
  - the VMM code should stay in generic
  - there is an amount of code still arch dependent
  - after stabilizing the ARM implementation we can make a common interface

FreeBSD

Steps I've taken (2)

- created a new `sys/arm/vmm`
- copied the VMM interface from `sys/amd64/vmm`
  - the VMM code should stay in generic
  - there is an amount of code still arch dependent
  - after stabilizing the ARM implementation we can make a common interface

- created some low-level routines for installing the exception vector for Hyp-mode
  - the most important entry is the `Hypervisor` one
  - it jumps there whenever `hyp` instruction is called or a VM raises an exception

How the Host-OS is making hypervisor calls?

- executes the `hyp` instruction
- first parameter indicates the address of a routine
- in Hyp-mode the code checks that the call came from the Host-OS

FreeBSD

Memory mapping

- Hyp-mode is basically another address space with its own mappings
- New translation level (Stage-2 translation) for VM isolation

Memory mapping

- Hyp-mode is basically another address space with its own mappings
- New translation level (Stage-2 translation) for VM isolation
- Issue: only LPAE is supported for both translations
- FreeBSD doesn't support LPAE and we cannot leverage on its memory management

LPAE support

- Implement LPAE support in the VMM code
- Support for 40bit PA
- 3-level pagetables support (other formats are available but I've simplified the implementation)

LPAE support

- Implement LPAE support in the VMM code
- Support for 40bit PA
- 3-level pagetables support (other formats are available but I've simplified the implementation)
- Issue: On 32-bit we don't have the DMAP mechanism (we need the virtual address of each entry to be able to write on it)

LPAE support

- Implement LPAE support in the VMM code
- Support for 40bit PA
- 3-level pagetables support (other formats are available but I've simplified the implementation)
- Issue: On 32-bit we don't have the DMAP mechanism (we need the virtual address of each entry to be able to write on it)
- Created a shadow pagetable for each level 1 and level 2 pagetables which have the VAs

Steps I've taken (3)

- Mapped the hypervisor code at the same address in Hyp-Mode and in Host-OS
  - all the pointers passed between modes needs to be consistent
  - note: the Hyp-mode works with the MMU enabled using a normal stage-1 translation using it's own pagetables

FreeBSD

Steps I've taken (3)

- Mapped the hypervisor code at the same address in Hyp-Mode and in Host-OS
  - all the pointers passed between modes needs to be consistent
  - note: the Hyp-mode works with the MMU enabled using a normal stage-1 translation using it's own pagetables
- Implement the low-level code which is doing context switching between the Host-OS and the VM
  - Save and restore the context (e.g. registers, co-proc registers)

FreeBSD

Steps I've taken (4)

- Copied the `libvmmapi`, `bhyveload` and `bhyve` code creating new user-space tools for ARM

Steps I've taken (4)

- Copied the `libvmmapi`, `bhyveload` and `bhyve` code creating new user-space tools for ARM
- Crafted `bhyveloadarm` to map a Guest-OS memory and load its image (binary)

Steps I've taken (4)

- Copied the `libvmmapi`, `bhyveload` and `bhyve` code creating new user-space tools for ARM
- Crafted `bhyveloadarm` to map a Guest-OS memory and load its image (binary)
- Implement MMIO emulation using traps in a Stage-2 translation

Steps I've taken (4)

- Copied the `libvmmapi`, `bhyveload` and `bhyve` code creating new user-space tools for ARM
- Crafted `bhyveloadarm` to map a Guest-OS memory and load its image (binary)
- Implement MMIO emulation using traps in a Stage-2 translation
- Implement the paravirtualized serial console

Steps I've taken (4)

- Copied the `libvmmapi`, `bhyveload` and `bhyve` code creating new user-space tools for ARM
- Crafted `bhyveloadarm` to map a Guest-OS memory and load its image (binary)
- Implement MMIO emulation using traps in a Stage-2 translation
- Implement the paravirtualized serial console
- Started virtualizing interrupts

Interrupt Controller Virtualization

- 2 components: Distributor and CPU Interface

Interrupt Controller Virtualization

- 2 components: Distributor and CPU Interface
- ARM provides a CPU Virtual Interface which can be used directly by the VM

FreeBSD

Interrupt Controller Virtualization

- 2 components: Distributor and CPU Interface
- ARM provides a CPU Virtual Interface which can be used directly by the VM
- One needs to virtualize the accesses to the distributor

FreeBSD

Interrupt Controller Virtualization

- 2 components: Distributor and CPU Interface
- ARM provides a CPU Virtual Interface which can be used directly by the VM
- One needs to virtualize the accesses to the distributor
- Current status:
  - mapped CPU Interface over the CPU Virtual Interface

FreeBSD

Interrupt Controller Virtualization

- 2 components: Distributor and CPU Interface
- ARM provides a CPU Virtual Interface which can be used directly by the VM
- One needs to virtualize the accesses to the distributor
- Current status:
  - mapped CPU Interface over the CPU Virtual Interface
  - create the Virtual GIC infrastructure in the VMM

FreeBSD

Interrupt Controller Virtualization

- 2 components: Distributor and CPU Interface
- ARM provides a CPU Virtual Interface which can be used directly by the VM
- One needs to virtualize the accesses to the distributor
- Current status:
  - mapped CPU Interface over the CPU Virtual Interface
  - create the Virtual GIC infrastructure in the VMM
  - register the Distributor accesses for in-kernel emulation

FreeBSD

Interrupt Controller Virtualization

- 2 components: Distributor and CPU Interface
- ARM provides a CPU Virtual Interface which can be used directly by the VM
- One needs to virtualize the accesses to the distributor
- Current status:
  - mapped CPU Interface over the CPU Virtual Interface
  - create the Virtual GIC infrastructure in the VMM
  - register the Distributor accesses for in-kernel emulation
  - the VM passes the GIC initialization and goes further but ends up with some spurious interrupts due

Interrupt Controller Virtualization

- 2 components: Distributor and CPU Interface
- ARM provides a CPU Virtual Interface which can be used directly by the VM
- One needs to virtualize the accesses to the distributor
- Current status:
  - mapped CPU Interface over the CPU Virtual Interface
  - create the Virtual GIC infrastructure in the VMM
  - register the Distributor accesses for in-kernel emulation
  - the VM passes the GIC initialization and goes further but ends up with some spurious interrupts due
  - still searching the cause of it (probably a mis configuration in the GICH_* registers)

FreeBSD

Timer virtualization

- For now we have two unused timers (SP804) that I've mapped directly to the guest

Timer virtualization

- For now we have two unused timers (SP804) that I've mapped directly to the guest
- Expose the generic timer to the guest

Development platform for bhyve ARM

- FastModels from ARM emulating an CortexA15 (XX evaluation days, needs license from ARM)

Development platform for bhyve ARM

- FastModels from ARM emulating an CortexA15 (XX evaluation days, needs license from ARM)
- Running bhyve ARM on a real hardware platform
- WIP - running bhyve ARM on Samsung Exynos 5250 - it get stuck in the crafted init code

Next steps

- Add SMP support to the VMM code (basically execute the init function all CPUs and add some locks)

FreeBSD

Next steps

- Add SMP support to the VMM code (basically execute the init function all CPUs and add some locks)
- Emulate devices (at least the serial console and a disk)

Next steps

- Add SMP support to the VMM code (basically execute the init function all CPUs and add some locks)
- Emulate devices (at least the serial console and a disk)
- Run multiple bhyve VMs

Next steps

- Add SMP support to the VMM code (basically execute the init function all CPUs and add some locks)
- Emulate devices (at least the serial console and a disk)
- Run multiple bhyve VMs
- Try to run Linux as a guest OS on bhyve ARM

Next steps

- Add SMP support to the VMM code (basically execute the init function all CPUs and add some locks)
- Emulate devices (at least the serial console and a disk)
- Run multiple bhyve VMs
- Try to run Linux as a guest OS on bhyve ARM
- Unify the interface of the VMM (amd64/arm)

Next steps

- Add SMP support to the VMM code (basically execute the init function all CPUs and add some locks)
- Emulate devices (at least the serial console and a disk)
- Run multiple bhyve VMs
- Try to run Linux as a guest OS on bhyve ARM
- Unify the interface of the VMM (amd64/arm)
- Porting bhyve ARM to an ARMv8 platform

Conclusions

- Porting bhyve on ARM showed that the VMM interface design almost fits our needs
- The VMM still has some arch dependent code
- Lack of the LPAE in the FreeBSD base (hard-wire memory for VM)
- Type-2 hypervisor needs special care on ARM (for now)

Thank you for your attention!

ask questions