Porting bhyve on ARM

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

FreeBSD

BSDCan 2016
University of Ottawa
Ottawa, Canada
June 10 – 11, 2016
About me

- University POLITEHNICA of Bucharest
  - PhD Student: virtualization on embedded devices
  - Teaching Assistant: operating systems, systems architecture, networks
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 (e.g. ATA emulation, NE2000 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
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
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

Other type-2 implementation - KVM

VirtualOpenSystems did the same thing
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
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
Steps I’ve taken

- boot FreeBSD on FastModels (Fixed Virtual Platform)
  - aprox. 2 weeks
  - different bugs in the FDT/OFW subsystem
Steps I’ve taken

▶ boot FreeBSD on FastModels (Fixed Virtual Platform)
  ▶ approx. 2 weeks
  ▶ different bugs in the FDT/OFW subsystem

▶ 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
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
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
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 page tables 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
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)
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

ARM provides a CPU Virtual Interface which can be used directly by the VM.

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 to lack of proper Distributor emulation handling.
Interrupt Controller Virtualization

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

- 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 to lack of proper Distributor emulation handling
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 to lack of proper Distributor emulation handling.
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

The VM passes the GIC initialization and goes further but ends up with some spurious interrupts due to lack of proper Distributor emulation handling.
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
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 to lack of proper Distributor emulation handling
Distributor emulation status

- Register the Distributor address range accesses for in-kernel emulation
- Created internal structure to retain the state of the distributor for each VM
Distributor emulation status

- Register the Distributor address range accesses for in-kernel emulation
- Created internal structure to retain the state of the distributor for each VM
- Basically configs of each interrupt (e.g. irq_enable, irq_active, irq_state, irq_conf)
- All reads and writes to Distributor registers are handled
- Need to populate the LR register accordingly to the state of the distributor
- Sync the saved state to the running state
Distributor emulation status

- Register the Distributor address range accesses for in-kernel emulation
- Created internal structure to retain the state of the distributor for each VM
- Basically configs of each interrupt (e.g. irq_enable, irq_active, irq_state, irq_conf)
- All reads and writes to Distributor registers are handled
Distributor emulation status

- Register the Distributor address range accesses for in-kernel emulation
- Created internal structure to retain the state of the distributor for each VM
- Basically configs of each interrupt (e.g. irq_enable, irq_active, irq_state, irq_conf)
- All reads and writes to Distributor registers are handled
- Need to populate the LR register accordingly to the state of the distributor

Sync the saved state to the running state
Distributor emulation status

- Register the Distributor address range accesses for in-kernel emulation
- Created internal structure to retain the state of the distributor for each VM
- Basically configs of each interrupt (e.g. irq_enable, irq_active, irq_state, irq_conf)
- All reads and writes to Distributor registers are handled
- Need to populate the LR register accordingly to the state of the distributor
- the LR register contains all the active interrupts that are signaled to the CPU Virtual Interface of the VM
- Sync the saved state to the running state
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
Running bhyve ARM on a real hardware platform

- Samsung Exynos 5250
- The code base was very old from April 2015 and the board wasn’t booting up
Running bhyve ARM on a real hardware platform

- Samsung Exynos 5250
- The code base was very old from April 2015 and the board wasn’t booting up
- Did a rebase with the current HEAD and our development repo
  - fix locore-v6.S integration of Andrew’s LEAVE_HYP and our hypervisor stub install method
  - fix the problems with vGIC introduced by INTRNG
Running bhyve ARM on a real hardware platform

- Samsung Exynos 5250
- The code base was very old from April 2015 and the board wasn’t booting up
- Did a rebase with the current HEAD and our development repo
  - fix locore-v6.S integration of Andrew’s LEAVE_HYP and our hypervisor stub install method
  - fix the problems with vGIC introduced by INTRNG
- Get the latest u-boot which left the board in Hyp-mode state
Running bhyve ARM on a real hardware platform

- Samsung Exynos 5250
- The code base was very old from April 2015 and the board wasn’t booting up
- Did a rebase with the current HEAD and our development repo
  - fix locore-v6.S integration of Andrew’s LEAVE_HYP and our hypervisor stub install method
  - fix the problems with vGIC introduced by INTRNG
- Get the latest u-boot which left the board in Hyp-mode state
- hvc instruction is causing an ”undefined instruction” exception
Running bhyve ARM on a real hardware platform

- Samsung Exynos 5250
- The code base was very old from April 2015 and the board wasn’t booting up
- Did a rebase with the current HEAD and our development repo
  - fix locore-v6.S integration of Andrew’s LEAVE_HYP and our hypervisor stub install method
  - fix the problems with vGIC introduced by INTRNG
- Get the latest u-boot which left the board in Hyp-mode state
- hvc instruction is causing an "undefined instruction" exception
- At some point in time, the board refused to power-up
Running bhyve ARM on a real hardware platform

▶ Samsung Exynos 5250
▶ The code base was very old from April 2015 and the board wasn’t booting up
▶ Did a rebase with the current HEAD and our development repo
  ▶ fix locore-v6.S integration of Andrew’s LEAVE_HYP and our hypervisor stub install method
  ▶ fix the problems with vGIC introduced by INTRNG
▶ Get the latest u-boot which left the board in Hyp-mode state
▶ hvc instruction is causing an ”undefined instruction” exception
▶ At some point in time, the board refused to power-up
▶ Talked with Andrew and Peter and decided to go with Cubieboard2 (AllWinner A20 SoC). It’s on its way to Romania...
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
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 - from ARMv8.1 you can run the Host OS in Hyp-mode)

Thank you for your attention!

*ask questions*