VIRTUALIZING IO THROUGH 
THE IO MEMORY MANAGEMENT UNIT (IOMMU)

ANDY KEGEL, PAUL BLINZER, ARKA BASU, MAGGIE CHAN

ASPLOS 2016
WHAT THIS TUTORIAL WILL AND WILL NOT COVER

Definition of “IO” or “Device” or “IO Device”:
- Traditional IO includes GPU for graphics, NIC, storage controller, USB controller, etc.
- New IO (accelerators) includes general-purpose computation on a GPU (GPGPU), encryption accelerators, digital signal processors, etc.

Two Parts in Virtualizing an IO Device

- **Device specific: Virtual instances of device**
  - Virtual functions and Physical function in devices (PCIe® SR-IOV, MR-IOV)

- **System defined: IO Memory Management Unit or IOMMU**
  - Virtualizing DMA accesses (Address Translation and Protection)
  - Virtualizing Interrupts (Interrupt Remapping and Virtualizing)
WHAT THIS TUTORIAL WILL AND WILL NOT COVER

Definition of “IO” or “Device” or “IO Device”:
- Traditional IO includes GPU for graphics, NIC, storage controller, USB controller, etc.
- New IO (accelerators) includes general-purpose computation on a GPU (GPGPU), encryption accelerators, digital signal processors, etc.

Two Parts in Virtualizing an IO Device
- System defined: IO Memory Management Unit or IOMMU
  - Virtualizing DMA accesses (Address Translation and Protection)
  - Virtualizing Interrupts (Interrupt Remapping and Virtualizing)
MOTIVATION: TRADITIONAL DMA BY IO

NO SYSTEM VIRTUALIZATION
MOTIVATION: TRADITIONAL DMA BY IO

NO SYSTEM VIRTUALIZATION

Virtual Addresses

Core

MMU

Core

MMU

IO Device

IO Device

Memory
MOTIVATION: TRADITIONAL DMA BY IO
NO SYSTEM VIRTUALIZATION

Virtual Addresses → Protection Check
MMU → Physical Addresses
Core

Physical Addresses
MMU
Core

Virtual Addresses
IO Device

Physical Addresses
Memory

IO Device

Protection Check
MOTIVATION: TRADITIONAL DMA BY IO

NO SYSTEM VIRTUALIZATION
MOTIVATION: TRADITIONAL DMA BY IO

NO SYSTEM VIRTUALIZATION

Device Driver

Core

Core

Virtual Addresses

Protection Check

MMU

MMU

Physical Addresses

IO Device

IO Device

Setup

Memory

Core

Core

Memory
MOTIVATION: TRADITIONAL DMA BY IO

NO SYSTEM VIRTUALIZATION

Device Driver

Core

Core

MMU

MMU

IO Device

IO Device

Virtual Addresses

Protection Check

Physical Addresses

Physical Addresses

Memory

Setup

DMA Request
MOTIVATION: TRADITIONAL DMA BY IO

NO SYSTEM VIRTUALIZATION
MOTIVATION: TRADITIONAL DMA BY IO

NO SYSTEM VIRTUALIZATION

Device Driver

Core

Virtual Addresses

Protection Check

MMU

Physical Addresses

Core

IO Device

Physical Addresses

Setup

DMA Request

IO Device

Memory

Wrong location

No protection from malicious devices

---> “DMA Attack” (e.g., FinSpy)

No protection from buggy device driver

Side channel attack – leak information
MOTIVATION: TRADITIONAL DMA BY IO

NO SYSTEM VIRTUALIZATION

Device Driver

Core

Core

Virtual Addresses

Protection Check

MMU

MMU

Physical Addresses

Physical Addresses

IO Device

IO Device

Physical Addresses

DMA Request

Setup

Wrong location

Memory

XM No protection from malicious devices

--> “DMA Attack” (e.g., FinSpy)

XM No protection from buggy device driver

XM Side channel attack – leak information

XM Needs hardware enforced memory protection

Device Driver
MOTIVATION: VIRTUAL MACHINES ARE TRENDING

Tremendous growth in virtualization in server

Efficient access to IO under virtualization is important

Source: IDC Server Virtualization, MCS 2012
BACKGROUND: TRANSLATIONS IN VIRTUALIZED SYSTEM

- Guest OS 0
- Guest OS 1
- Hypervisor (a.k.a. VMM)
- Hardware – CPU, Memory, IO
BACKGROUND: TRANSLATIONS IN VIRTUALIZED SYSTEM

Guest Virtual Address (GVA)
- Guest Applications
- Guest OS 0
- Guest Applications
- Guest OS 1
- Hypervisor (a.k.a. VMM)
- Hardware – CPU, Memory, IO
BACKGROUND: TRANSLATIONS IN VIRTUALIZED SYSTEM

Guest Virtual Address (GVA) -> Guest OS 0

Guest Virtual Address (GVA) -> Guest OS 1

Hypervisor (a.k.a. VMM)

Guest Applications

Managed by Guest OS

Hardware – CPU, Memory, IO
BACKGROUND: TRANSLATIONS IN VIRTUALIZED SYSTEM

- Guest Virtual Address (GVA)
- Guest Physical Address (GPA)
- System Physical Address (SPA)
- Hypervisor (a.k.a. VMM)
- Hardware – CPU, Memory, IO

Guest Applications

Guest OS 0

Guest OS 1

Managed by
Guest OS

Managed by
VMM
BACKGROUND: TRANSLATIONS IN VIRTUALIZED SYSTEM

Guest Virtual Address (GVA) -> Guest OS 0

Guest Physical Address (GPA) -> Hypervisor (a.k.a. VMM)

System Physical Address (SPA) -> Hardware – CPU, Memory, IO

Guest Applications

Guest OS 0

Guest Applications

Guest OS 1

Isolation across Guest OS => No access to (system) physical address from Guest OS
MOTIVATION: TRADITIONAL DMA IN VIRTUAL MACHINES
VIRTUALIZED SYSTEM

Core

MMU

Core

MMU

IO Device

IO Device

Memory

*SPA == “Physical Address”
MOTIVATION: TRADITIONAL DMA IN VIRTUAL MACHINES
VIRTUALIZED SYSTEM

Guest OS 0
Core
MMU
Memory

Guest OS 1
Core
MMU
Memory

VMM

IO Device

GVA
GPA
SPA

*SPA == “Physical Address”
MOTIVATION: TRADITIONAL DMA IN VIRTUAL MACHINES

VIRTUALIZED SYSTEM

- Guest OS 0
- Guest OS 1
- Device Driver
- Core
- Core
- IO Device
- IO Device
- VMM
- MMU
- MMU
- Memory

GVA

GPA

SPA

No access to Physical Address

Setup

*SPA == “Physical Address”
MOTIVATION: TRADITIONAL DMA IN VIRTUAL MACHINES
VIRTUALIZED SYSTEM

Every DMA operation mediated by VMM

Device Driver

Memory

SPA == “Physical Address”
MOTIVATION: TRADITIONAL DMA IN VIRTUAL MACHINES

VIRTUALIZED SYSTEM

GVA

Core

Guest OS 0

MMU

SPA

Core

Guest OS 1

MMU

DMA Operation

Every DMA operation mediated by VMM

Setup

Setup

Physical Addresses

Device Driver

IO Device

IO Device

Memory

*SPA == “Physical Address”
MOTIVATION: TRADITIONAL DMA IN VIRTUAL MACHINES
VIRTUALIZED SYSTEM

Every DMA operation mediated by VMM
→ Often ~30% performance overhead

*SPA == “Physical Address”
MOTIVATION: TRADITIONAL DMA IN VIRTUAL MACHINES
VIRTUALIZED SYSTEM

- **GVA**: Guest OS 0, Core, MMU, Memory
- **GPA**: Guest OS 1, Core, MMU, Memory
- **SPA**: VMM, Device Driver, IO Device, DMA Operation

Every DMA operation mediated by VMM
→ Often ~30% performance overhead

*SPA == “Physical Address”*
INTRODUCTION OF IOMMU: THE LOGICAL VIEW

Core → MMU → Memory → Core
Core → MMU → Memory → Core
IO Device → MMU → Memory → IO Device
IO Device → MMU → Memory → IO Device
INTRODUCTION OF IOMMU: THE LOGICAL VIEW

Key capabilities:
1. Memory protection for DMA
2. Virtual address translation for DMA

IOMMU Driver
Sets up IOMMU hardware

Core

MMU

Memory

IO Device

IOMMU

Hardware that intercepts DMA transactions

Core

MMU

IO Device
MOTIVATION: TRADITIONAL IO INTERRUPT
NON-VIRTUALIZED SYSTEM

Core

MMU

Core

IO Device

IO Device

Memory
MOTIVATION: TRADITIONAL IO INTERRUPT
NON-VIRTUALIZED SYSTEM

Device Driver

Core

Core

MMU

MMU

IO Device

IO Device

Memory

Setup

IRQ # + Core id
MOTIVATION: TRADITIONAL IO INTERRUPT
NON-VIRTUALIZED SYSTEM

Device Driver

Core
APIC

MMU

Core
APIC

MMU

IO Device

IO Device

Memory

Setup

IRQ # + Core id

IRQ #
MOTIVATION: TRADITIONAL IO INTERRUPT
VIRTUALIZED SYSTEM

- Core
- Guest OS
- VMM
- MMU
- IO Device
- IO Device
- Memory

VMM (Virtual Machine Monitor) manages the communication between the guest operating systems and the hardware.
MOTIVATION: TRADITIONAL IO INTERRUPT
VIRTUALIZED SYSTEM

- Guest OS
- Core
- MMU
- VMM
- IO Device
- IRQ 
- Memory

Set up

33 IOMMU TUTORIAL @ ASPLOS | 3rd APRIL 2016
MOTIVATION: TRADITIONAL IO INTERRUPT
VIRTUALIZED SYSTEM

Guest OS 0
Core
VMM
MMU
Guest OS migration
IO Device
Setup
Memory

IRQ # + Core i
MOTIVATION: TRADITIONAL IO INTERRUPT VIRTUALIZED SYSTEM

Guest OS 0

Core

VMM

MMU

Memory

Guest OS migration

IO Device

Setup

IO Device

IRQ # + Core i
**MOTIVATION: TRADITIONAL IO INTERRUPT**

**VIRTUALIZED SYSTEM**

- **Guest OS 0**
- **Core**
- **VMM**
- **MMU**
- **Inter-Process Interrupt**
- **Memory**
- **IO Device**
- **Setup**
- **Guest OS migration**
- **IRQ # + Core i**
MOTIVATION: TRADITIONAL IO INTERRUPT
VIRTUALIZED SYSTEM

- Extraneous IPI adds overheads
- Each extra interrupt can add 5-10K cycles
- Needs dynamic remapping of interrupts

**MOTIVATION**

Virtualized System

- Guest OS
- Core
- MMU

- IO Device
- VMM

**Setup**

- IRQ # + Core i

**Inter-Process Interrupt**

- Memory

**Guest OS migration**
MOTIVATION: TRADITIONAL IO INTERRUPT
VIRTUALIZED SYSTEM

Diagram showing the traditional approach to I/O interrupts in a virtualized system, with components such as VMM, MMUs, cores, guest OS, IO devices, and memory.
MOTIVATION: TRADITIONAL IO INTERRUPT
VIRTUALIZED SYSTEM

Performance overheads VMM exits on each interrupt
MOTIVATION: TRADITIONAL IO INTERRUPT
VIRTUALIZED SYSTEM

Core

Guest OS 0

VMM

MMU

IO Device

IO Device

Setup

Guest OS de-scheduled

 IRQ # + Core i

Memory

Performance overheads VMM exits on each interrupt
MOTIVATION: TRADITIONAL IO INTERRUPT
VIRTUALIZED SYSTEM

Core

Guest OS

IO Device

Guest OS de-scheduled

Setup

IRQ # + Core i

VMM

MMU

Memory

Guest OS de-scheduled

Performance overheads VMM exits on each interrupt

Unnecessary VMM wakeup
MOTIVATION: TRADITIONAL IO INTERRUPT
VIRTUALIZED SYSTEM

Need to virtualize interrupt:
- Direct interrupt delivery to guest OS and temporary queueing

Setup

Guest OS de-scheduled

Performance overheads VMM exits on each interrupt

Unnecessary VMM wakeup

Memory

Core

Guest OS

IO Device

IO Device

Core

Core

VMM

MMU

MMU

IRQ # + Core i

Need to virtualize interrupt:
- Direct interrupt delivery to guest OS and temporary queueing
INTRODUCTION OF IOMMU: THE LOGICAL VIEW
ADDING INTERRUPT HANDLING CAPABILITY

IOMMU Driver
Sets up IOMMU hardware

Core
Core

MMU
MMU

IO Device
IO Device

IOMMU

Hardware that intercepts DMA transactions

Key capabilities:
1. Memory protection for DMA
2. Virtual address translation for DMA

Memory
INTRODUCTION OF IOMMU: THE LOGICAL VIEW

ADDING INTERRUPT HANDLING CAPABILITY

IOMMU Driver

Sets up IOMMU hardware

Core

Core

MMU

MMU

IO Device

IO Device

IOMMU

Hardware that intercepts DMA transactions and interrupts

Key capabilities:
1. Memory protection for DMA
2. Virtual address translation for DMA
3. Interrupt remapping and virtualization

Memory
MOTIVATION: EMERGENCE OF HETEROGENEOUS SYSTEMS
HETEROGENEOUS SYSTEM ARCHITECTURE (HSA)
MOTIVATION: EMERGENCE OF HETEROGENEOUS SYSTEMS

HETEROGENEOUS SYSTEM ARCHITECTURE (HSA)

- Core
- MMU
- Memory
- Core
- MMU
- GPU
- IO Device

Memory
MOTIVATION: EMERGENCE OF HETEROGENEOUS SYSTEMS

HETEROGENEOUS SYSTEM ARCHITECTURE (HSA)

Shared virtual addressing is key to ease of programming

Core

MMU

Memory

Core

MMU

GPU

IO Device

virtual addressing is key to ease of programming
MOTIVATION: EMERGENCE OF HETEROGENEOUS SYSTEMS

HETEROGENEOUS SYSTEM ARCHITECTURE (HSA)

Shared virtual addressing is key to ease of programming

Core

IO Device

MMU

Memory
MOTIVATION: EMERGENCE OF HETEROGENEOUS SYSTEMS

HETEROGENEOUS SYSTEM ARCHITECTURE (HSA)

Shared virtual addressing is key to ease of programming

"Pointer-is-a-Pointer" across CPU and devices

Memory

Core

MMU

IO Device
MOTIVATION: EMERGENCE OF HETEROGENEOUS SYSTEMS
HETEROGENEOUS SYSTEM ARCHITECTURE (HSA)

Shared virtual addressing is key to ease of programming

“Pointer-is-a-Pointer” across CPU and devices

IO needs to share CPU page table*

*Data Structure that keeps VA to PA mapping
INTRODUCTION OF IOMMU: THE LOGICAL VIEW
ADDING ABILITY TO SHARE ADDRESS SPACE IN HETEROGENEOUS SYSTEM

Key capabilities:
1. Memory protection for DMA
2. Virtual address translation for DMA
3. Interrupt remapping and virtualization

Hardware that intercepts DMA transactions and interrupts

Sets up IOMMU hardware
INTRODUCTION OF IOMMU: THE LOGICAL VIEW
ADDING ABILITY TO SHARE ADDRESS SPACE IN HETEROGENEOUS SYSTEM

Key capabilities:
1. Memory protection for DMA
2. Virtual address translation for DMA
3. Interrupt remapping and virtualization
4. IO can share CPU page tables
INTRODUCTION OF IOMMU: (TYPICAL) PHYSICAL VIEW
IOMMU IS PART OF PROCESSOR COMPLEX

Processor /Chip

Core

Core

MMU

MMU

Memory Controller

Memory

IO Device

IO Device

IO Device

IOMMU

Root Complex/ “IOHUB”

Interconnect
IOMMU FROM THE PERSPECTIVE OF DEVICE (PCIE® SPEC)

Memory

Translation Agent

Addr. Translation and Protection Table

Root Complex (RC)

Root Integrated Endpoint

Root Port

Device

ATC

Switch

ATC – Address Translation Cache
IOMMU FROM THE PERSPECTIVE OF DEVICE (PCIE® SPEC)

IOMMU → Translation Agent and uses the Address Translation and Protection Table

ATC – Address Translation Cache
### COMPARING CPU MMU AND IOMMU

<table>
<thead>
<tr>
<th></th>
<th>CPU MMU</th>
<th>IOMMU</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Address Translation</strong></td>
<td>VA $\rightarrow$ PA and GVA $\rightarrow$ GPA $\rightarrow$ SPA</td>
<td>VA $\rightarrow$ PA and GVA $\rightarrow$ GPA $\rightarrow$ SPA</td>
</tr>
<tr>
<td><strong>Memory Protection</strong></td>
<td>Read/Write etc.</td>
<td>Read/Write etc.</td>
</tr>
<tr>
<td><strong>Interrupt Handling</strong></td>
<td>No</td>
<td>Remapping and Virtualization Support</td>
</tr>
<tr>
<td><strong>Parallelism</strong></td>
<td>Mostly Single Threaded</td>
<td>Highly Multithreaded</td>
</tr>
<tr>
<td><strong>Page Faults, Events, etc.</strong></td>
<td>Synchronous Handling</td>
<td>Asynchronous Handling</td>
</tr>
</tbody>
</table>
HISTORY
A SIMPLIFIED VIEW

V1, c. 2004
Technology created to translate and vet memory accesses by peripherals, replacing software

V1.2, c. 2006
Interrupt remapping added for IO virtualization

V2, c. 2008
Nested paging, interrupt virtualization, and improved management features added

V3, c. 2010
Features added for full heterogeneous computing and further efficiencies

Whither next?
IOMMU TECHNOLOGY FAMILIES

REFERENCES

AMD IOMMU®
- IO Memory Management Unit

Intel VT-d®
- Virtualization Technology for Directed IO

ARM SMM®
- System Memory Management Unit

IBM CAPI®
- Coherent Accelerator Processor Interface
AGENDA

USE CASES & DEMO STRATION
Where can IOMMU help?

INTERNALS
How does IOMMU work?

RESEARCH
Research Opportunities and Tools
FIVE USE CASES OF IOMMU

LEGACY I/O
- Supporting legacy devices –
  Extending DMA “beyond reach”

SECURITY AND PROTECTION
- Preventing uncontrolled memory access

SECURE BOOT
- Enforcing secure boot

DIRECT I/O DEVICES
- Secure and efficient IO from Guest OS

HETEROGENEOUS COMPUTING
- Enabling shared virtual memory
SUPPORTING LEGACY DEVICES
HOW CAN AN IOMMU HELP?

Many 32-bit DMA devices operate in a 64-bit system
- Older PCI cards (through PCI-PCIe bridges), special-purpose controllers, parallel ports (IEEE-1284), ...
SUPPORTING LEGACY DEVICES
HOW CAN AN IOMMU HELP?

Many 32-bit DMA devices operate in a 64-bit system
- Older PCI cards (through PCI-PCIe bridges), special-purpose controllers, parallel ports (IEEE-1284), ...
SUPPORTING LEGACY DEVICES
HOW CAN AN IOMMU HELP?

- Many 32-bit DMA devices operate in a 64-bit system
  - Older PCI cards (through PCI-PCIe bridges), special-purpose controllers, parallel ports (IEEE-1284), ...

- SW Solution: Bounce buffers
  - Device does DMA to a region in 32bit physical address, CPU copies data from buffer to the final destination
SUPPORTING LEGACY DEVICES

HOW CAN AN IOMMU HELP?

- Many 32-bit DMA devices operate in a 64-bit system
  - Older PCI cards (through PCI-PCIe bridges), special-purpose controllers, parallel ports (IEEE-1284), ...

- SW Solution: Bounce buffers
  - Device does DMA to a region in 32bit physical address, CPU copies data from buffer to the final destination
SUPPORTING LEGACY DEVICES
HOW CAN AN IOMMU HELP?

- Many 32-bit DMA devices operate in a 64-bit system
  - Older PCI cards (through PCI-PCIe bridges), special-purpose controllers, parallel ports (IEEE-1284), ...

- SW Solution: Bounce buffers
  - Device does DMA to a region in 32bit physical address, CPU copies data from buffer to the final destination
Many 32-bit DMA devices operate in a 64-bit system
- Older PCI cards (through PCI-PCIe bridges), special-purpose controllers, parallel ports (IEEE-1284), ...

SW Solution: Bounce buffers
- Device does DMA to a region in 32-bit physical address, CPU copies data from buffer to the final destination
SUPPORTING LEGACY DEVICES
HOW CAN AN IOMMU HELP?

- Many 32-bit DMA devices operate in a 64-bit system
  - Older PCI cards (through PCI-PCIe bridges), special-purpose controllers, parallel ports (IEEE-1284), ...

- SW Solution: Bounce buffers
  - Device does DMA to a region in 32bit physical address, CPU copies data from buffer to the final destination
### SUPPORTING LEGACY DEVICES

#### HOW CAN AN IOMMU HELP?

- Many 32-bit DMA devices operate in a 64-bit system
  - Older PCI cards (through PCI-PCIe bridges), special-purpose controllers, parallel ports (IEEE-1284), ...

- **SW Solution: Bounce buffers**
  - Device does DMA to a region in 32bit physical address, CPU copies data from buffer to the final destination
  - Slow, needs SW synchronization, ties up CPU core
SUPPORTING LEGACY DEVICES

HOW CAN AN IOMMU HELP?

Many 32bit DMA devices operate in a 64bit system
   - older PCI cards (through PCI-PCIe bridges), special-purpose controllers, parallel ports (IEEE-1284), ...
SUPPORTING LEGACY DEVICES

HOW CAN AN IOMMU HELP?

- Many 32-bit DMA devices operate in a 64-bit system
  - older PCI cards (through PCI-PCIe bridges), special-purpose controllers, parallel ports (IEEE-1284), ...

- Better solution: IOMMU remaps 32-bit device physical address to system physical address beyond 32-bit
SUPPORTING LEGACY DEVICES

HOW CAN AN IOMMU HELP?

- Many 32bit DMA devices operate in a 64bit system
  - older PCI cards (through PCI-PCIe bridges), special-purpose
    controllers, parallel ports (IEEE-1284), ...

- Better solution: IOMMU remaps 32bit device physical
  address to system physical address beyond 32bit
SUPPORTING LEGACY DEVICES
HOW CAN AN IOMMU HELP?

- Many 32bit DMA devices operate in a 64bit system
  - older PCI cards (through PCI-PCIe bridges), special-purpose controllers, parallel ports (IEEE-1284), ...
- Better solution: IOMMU remaps 32bit device physical address to system physical address beyond 32bit

![IOMMU Diagram](image-url)
SUPPORTING LEGACY DEVICES

HOW CAN AN IOMMU HELP?

- Many 32bit DMA devices operate in a 64bit system
  - older PCI cards (through PCI-PCIe bridges), special-purpose controllers, parallel ports (IEEE-1284), ...

- Better solution: IOMMU remaps 32bit device physical address to system physical address beyond 32bit
SUPPORTING LEGACY DEVICES
HOW CAN AN IOMMU HELP?

- Many 32bit DMA devices operate in a 64bit system
  - older PCI cards (through PCI-PCIe bridges), special-purpose controllers, parallel ports (IEEE-1284), ...

- Better solution: IOMMU remaps 32bit device physical address to system physical address beyond 32bit
  - DMA goes directly into 64bit memory
  - No CPU transfer
  - More efficient
SUPPORTING LEGACY DEVICES
HOW CAN AN IOMMU HELP?

- Many 32bit DMA devices operate in a 64bit system
  - older PCI cards (through PCI-PCIe bridges), special-purpose controllers, parallel ports (IEEE-1284), ...

- Better solution: IOMMU remaps 32bit device physical address to system physical address beyond 32bit
  - DMA goes directly into 64bit memory
  - No CPU transfer
  - More efficient

- Linux: DMA redirect feature
IOMMU USECASE: SECURITY AND PROTECTION SECURITY BOOT
The traditional IOMMU use:

DMA devices use physical addresses on the system bus to read and write memory based on SW driver or OS instructions.
SECURITY AND PROTECTION
THE TRADITIONAL IOMMU USE

DMA devices use physical addresses on the system bus to read and write memory based on SW driver or OS instructions.
SECURITY AND PROTECTION
THE TRADITIONAL IOMMU USE

- DMA devices use physical addresses on the system bus to read and write memory based on SW driver or OS instructions
- SW bugs or attacks by malicious applications could access and modify important OS data (OS security policy, passwords, ...)
  - Without OS able to detect or prevent the access as it can for CPU
  - Latent problem until it shows unexpectedly possibly much later
SECURITY AND PROTECTION
THE TRADITIONAL IOMMU USE

- DMA devices use physical addresses on the system bus to read and write memory based on SW driver or OS instructions
- SW bugs or attacks by malicious applications could access and modify important OS data (OS security policy, passwords,...)
  - Without OS able to detect or prevent the access as it can for CPU
  - Latent problem until it shows unexpectedly possibly much later

Device

I/O buffer

Passwords, Critical data

Physical Memory
SECURITY AND PROTECTION
THE TRADITIONAL IOMMU USE

- DMA devices use physical addresses on the system bus to read and write memory based on SW driver or OS instructions
- SW bugs or attacks by malicious applications could access and modify important OS data (OS security policy, passwords, ...)
  - Without OS able to detect or prevent the access as it can for CPU
  - Latent problem until it shows unexpectedly possibly much later
- This affects system stability, if just the right data is hit
  - “Heisenbugs” are sometimes caused by bugs in system drivers
- Or it allows malicious driver attacks to take over the system

![Diagram showing device, I/O buffer, critical data, and passwords with physical memory]

Passwords,
Critical data
I/O buffer
Device
SECURITY AND PROTECTION

THE TRADITIONAL IOMMU USE

- DMA devices assert physical addresses on the system bus to read and write memory based on SW driver or OS settings.
- SW bugs or attacks by malicious applications could access and modify important data (OS security policy, passwords, ...)

Diagram:
- Physical Memory
- Device
- IOMMU
- Range check
- I/O buffer
- Passwords, critical data
DMA devices assert physical addresses on the system bus to read and write memory based on SW driver or OS settings.

Sw bugs or attacks by malicious applications could access and modify important data (OS security policy, passwords,...)

The IOMMU allows OS to enforce DMA access policy for any DMA capable device accessing physical memory:
- Memory state important to stability/security
- If access occurs, OS gets notified and can shut the device & driver down and notifies the user or administrator.
THE TRADITIONAL IOMMU USE

- DMA devices assert physical addresses on the system bus to read and write memory based on SW driver or OS settings.
- SW bugs or attacks by malicious applications could access and modify important data (OS security policy, passwords,...)
- The IOMMU allows OS to enforce DMA access policy for any DMA capable device accessing physical memory:
  - Memory state important to stability/security
  - If access occurs, OS gets notified and can shut the device & driver down and notifies the user or administrator.
SECURITY AND PROTECTION
THE TRADITIONAL IOMMU USE

- DMA devices assert physical addresses on the system bus to read and write memory based on SW driver or OS settings
- SW bugs or attacks by malicious applications could access and modify important data (OS security policy, passwords,...)
- The IOMMU allows OS to enforce DMA access policy for any DMA capable device accessing physical memory
  - Memory state important to stability/security
  - If access occurs, OS gets notified and can shut the device & driver down and notifies the user or administrator
SECURITY AND PROTECTION
THE TRADITIONAL IOMMU USE

- DMA devices assert physical addresses on the system bus to read and write memory based on SW driver or OS settings
- SW bugs or attacks by malicious applications could access and modify important data (OS security policy, passwords,...)
- The IOMMU allows OS to enforce DMA access policy for any DMA capable device accessing physical memory
  - Memory state important to stability/security
  - If access occurs, OS gets notified and can shut the device & driver down and notifies the user or administrator
SECURITY AND PROTECTION
THE TRADITIONAL IOMMU USE

- DMA devices assert physical addresses on the system bus to read and write memory based on SW driver or OS settings.
- SW bugs or attacks by malicious applications could access and modify important data (OS security policy, passwords,...).
- The IOMMU allows OS to enforce DMA access policy for any DMA capable device accessing physical memory.
  - Memory state important to stability/security.
  - If access occurs, OS gets notified and can shut the device & driver down and notifies the user or administrator.
SECURE BOOT
YET ANOTHER USE FOR AN IOMMU

- Ensuring that a system is not doing more than it’s supposed to
  - e.g., being part of a botnet, provide banking data or other personal info to impersonators or other attackers
  - The earliest time for attack and defense is at firmware startup
  - From there critical memory regions are protected from invalid access
SECURE BOOT
YEAT ANOTHER USE FOR AN IOMMU

- Ensuring that a system is not doing more than it’s supposed to
  - e.g., being part of a botnet, provide banking data or other personal info to impersonators or other attackers
  - The earliest time for attack and defense is at firmware startup
  - From there critical memory regions are protected from invalid access

- The Secure Boot architecture ensures that no non-vetted OS kernel code runs on the system, changing critical settings
SECURE BOOT
YET ANOTHER USE FOR AN IOMMU

- Ensuring that a system is not doing more than it’s supposed to
  - e.g., being part of a botnet, provide banking data or other personal info to impersonators or other attackers
  - The earliest time for attack and defense is at firmware startup
  - From there critical memory regions are protected from invalid access
- The Secure Boot architecture ensures that no non-vetted OS kernel code runs on the system, changing critical settings
- Some I/O devices can issue DMA requests to system memory directly, without OS or Firmware intervention
  - e.g., 1394/Firewire, network cards, as part of network boot
  - That allows attacks to modify memory before even the OS has a chance to protect against the attacks
SECURE BOOT
YET ANOTHER USE FOR AN IOMMU

- Ensuring that a system is not doing more than it’s supposed to
  - e.g., being part of a botnet, provide banking data or other personal info to impersonators or other attackers
  - The earliest time for attack and defense is at firmware startup
  - From there critical memory regions are protected from invalid access

- The Secure Boot architecture ensures that no non-vetted OS kernel code runs on the system, changing critical settings

- Some I/O devices can issue DMA requests to system memory directly, without OS or Firmware intervention
  - e.g., 1394/Firewire, network cards, as part of network boot
  - That allows attacks to modify memory before even the OS has a chance to protect against the attacks

- As outlined earlier, using the IOMMU prevents DMA access to important memory regions
IOMMU USECASE: EFFICIENT IO IN VIRTUALIZED ENVIRONMENT
BACKGROUND: TRADITIONAL DMA BY IO
(NO SYSTEM VIRTUALIZATION)
BACKGROUND: TRADITIONAL DMA BY IO
(NO SYSTEM VIRTUALIZATION)
BACKGROUND: TRADITIONAL DMA BY IO
(NO SYSTEM VIRTUALIZATION)
BACKGROUND: TRADITIONAL DMA BY IO
(NO SYSTEM VIRTUALIZATION)

Virtual Addresses

Core

Core

Protection Check

MMU

MMU

Physical Addresses

Device Driver

IO Device

IO Device

Memory

Device Driver

Virtual Addresses

Physical Addresses

MMU

Memory

IO Device
BACKGROUND: TRADITIONAL DMA BY IO
(NO SYSTEM VIRTUALIZATION)

Virtual Addresses

Physical Addresses

Device Driver

Protection Check

Setup

Core

Core

MMU

MMU

IO Device

IO Device

Memory
BACKGROUND: TRADITIONAL DMA BY IO
(NO SYSTEM VIRTUALIZATION)
BACKGROUND: TRADITIONAL DMA BY IO
(NO SYSTEM VIRTUALIZATION)

- Core
- Core
- IO Device
- IO Device
- MMU
- MMU
- Device Driver
- Memory

Virtual Addresses → MMU → Core → IO Device
Physical Addresses → MMU → Core → IO Device
Protection Check

Setup:
 DMA Request

Virtual Addresses → Device Driver → IO Device
BACKGROUND: TRADITIONAL DMA BY IO
(NO SYSTEM VIRTUALIZATION)

Device drivers must program the true system physical memory address
No protection from SW or hardware bugs in I/O devices and drivers
system crash by writing wrong memory
No protection from potentially malicious driver or system SW attacks
VIRTUALIZATION OF A SYSTEM IN SOFTWARE
IT HAS TO LOOK REAL TO AN OPERATING SYSTEM

- Each OS assumes full access to the platform hardware
  - Memory, Interrupts, Devices, CPU cores, etc.
VIRTUALIZATION OF A SYSTEM IN SOFTWARE
IT HAS TO LOOK REAL TO AN OPERATING SYSTEM

- Each OS assumes full access to the platform hardware
  - Memory, Interrupts, Devices, CPU cores, etc.

- A Virtual Machine Manager (VMM) or Hypervisor (HV) is tasked to manage the physical hardware and define a “virtual machine” (VM) that represents the resources an OS expects to find in the system.
VIRTUALIZATION OF A SYSTEM IN SOFTWARE
IT HAS TO LOOK REAL TO AN OPERATING SYSTEM

- Each OS assumes full access to the platform hardware
  - Memory, Interrupts, Devices, CPU cores, etc.

- A Virtual Machine Manager (VMM) or Hypervisor (HV) is tasked to manage
  the physical hardware and define a “virtual machine” (VM) that represents
  the resources an OS expects to find in the system.
VIRTUALIZATION OF A SYSTEM IN SOFTWARE
IT HAS TO LOOK REAL TO AN OPERATING SYSTEM

- Each OS assumes full access to the platform hardware
  - Memory, Interrupts, Devices, CPU cores, etc.
- A Virtual Machine Manager (VMM) or Hypervisor (HV) is tasked to manage
  the physical hardware and define a “virtual machine” (VM) that represents
  the resources an OS expects to find in the system.
- Use cases:
VIRTUALIZATION OF A SYSTEM IN SOFTWARE
IT HAS TO LOOK REAL TO AN OPERATING SYSTEM

- Each OS assumes full access to the platform hardware
  - Memory, Interrupts, Devices, CPU cores, etc.

- A Virtual Machine Manager (VMM) or Hypervisor (HV) is tasked to manage the physical hardware and define a “virtual machine” (VM) that represents the resources an OS expects to find in the system.

- Use cases:
  - System consolidation
VIRTUALIZATION OF A SYSTEM IN SOFTWARE
IT HAS TO LOOK REAL TO AN OPERATING SYSTEM

- Each OS assumes full access to the platform hardware
  - Memory, Interrupts, Devices, CPU cores, etc.
- A Virtual Machine Manager (VMM) or Hypervisor (HV) is tasked to manage the physical hardware and define a “virtual machine” (VM) that represents the resources an OS expects to find in the system
- Use cases:
  - System consolidation
  - OS/application compatibility
VIRTUALIZATION OF A SYSTEM IN SOFTWARE
IT HAS TO LOOK REAL TO AN OPERATING SYSTEM

- Each OS assumes full access to the platform hardware
  - Memory, Interrupts, Devices, CPU cores, etc.

- A Virtual Machine Manager (VMM) or Hypervisor (HV) is tasked to manage the physical hardware and define a “virtual machine” (VM) that represents the resources an OS expects to find in the system

- Use cases:
  - System consolidation
  - OS/application compatibility
  - Security / Stability
VIRTUALIZATION OF A SYSTEM IN SOFTWARE
IT HAS TO LOOK REAL TO AN OPERATING SYSTEM

- Each OS assumes full access to the platform hardware
  - Memory, Interrupts, Devices, CPU cores, etc.

- A Virtual Machine Manager (VMM) or Hypervisor (HV) is tasked to manage
  the physical hardware and define a “virtual machine” (VM) that represents
  the resources an OS expects to find in the system

- Use cases:
  - System consolidation
  - OS/application compatibility
  - Security / Stability
  - Cloud Infrastructure
VIRTUALIZATION OF A SYSTEM

Most CPUs today have support for system virtualization

- Nested page tables (HV & OS levels), allow VMM/HV to assign and manage system memory and interrupts to Virtual Machines
Most CPUs today have support for system virtualization
- Nested page tables (HV & OS levels), allow VMM/HV to assign and manage system memory and interrupts to Virtual Machines

I/O devices are typically managed by HV/VMM software, either by...
VIRTUALIZATION OF A SYSTEM

† Most CPUs today have support for system virtualization
  – Nested page tables (HV & OS levels), allow VMM/HV to assign and manage system memory and interrupts to Virtual Machines

† I/O devices are typically managed by HV/VMM software, either by...

Para-Virtualization

<table>
<thead>
<tr>
<th>Guest device driver uses HV “hypercalls”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hypervisor manages HW operation (DMA)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Hypervisor SW validates and redirects I/O requests from Guest OS (overhead, slow)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Hypervisor arbitrates and schedules requests from multiple guest OS, allows VM migration</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Most common operation for today’s virtualization Software</td>
</tr>
<tr>
<td>Works well for CPU-heavy workloads</td>
</tr>
<tr>
<td>I/O, graphics or compute-heavy workloads</td>
</tr>
</tbody>
</table>
VIRTUALIZATION OF A SYSTEM

- Most CPUs today have support for system virtualization
  - Nested page tables (HV & OS levels), allow VMM/HV to assign and manage system memory and interrupts to Virtual Machines

- I/O devices are typically managed by HV/VMM software, either by...

<table>
<thead>
<tr>
<th>Para-Virtualization</th>
<th>Direct-Mapped Device &amp; SR-IOV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guest device driver uses HV “hypercalls” Hypervisor manages HW operation (DMA)</td>
<td>Device function is mapped to guest OS  Guest OS uses native HW drivers</td>
</tr>
<tr>
<td>Hypervisor SW validates and redirects I/O requests from Guest OS (overhead, slow)</td>
<td>Physical Device DMA must be limited and redirected by Hypervisor (via IOMMU),</td>
</tr>
<tr>
<td>Hypervisor arbitrates and schedules requests from multiple guest OS, allows VM migration</td>
<td>One device function per guest OS, physical memory must be committed</td>
</tr>
<tr>
<td>Most common operation for today’s virtualization Software Works well for CPU-heavy workloads I/O, graphics or compute-heavy workloads</td>
<td>I/O device must be resettable by HV when guest error puts it in undefined state  SR-IOV is a variant of direct mapped  I/O device provides 1 - n “virtual” devices in HW (PCI-SIG standard)</td>
</tr>
</tbody>
</table>
EFFICIENT I/O VIRTUALIZATION
HARDWARE IMPLEMENTED TECHNIQUE THROUGH IOMMU

IOMMU validates DMA accesses and validates device interrupts
EFFICIENT IO VIRTUALIZATION WITH IOMMU
WHAT ARE THE BENEFITS?

Using the IOMMU allows a Hypervisor to assign a physical device exclusively to a Guest VM without danger of memory corruption to other VMs.
EFFICIENT IO VIRTUALIZATION WITH IOMMU
WHAT ARE THE BENEFITS?

- Using the IOMMU allows a Hypervisor to assign a physical device exclusively to a Guest VM without danger of memory corruption to other VMs
  - Beneficial if one VM requires near native performance
EFFICIENT IO VIRTUALIZATION WITH IOMMU
WHAT ARE THE BENEFITS?

- Using the IOMMU allows a Hypervisor to assign a physical device exclusively to a Guest VM without danger of memory corruption to other VMs
  - Beneficial if one VM requires near native performance
  - Or if OS needs to be “sandboxed” (because of suspected malware)
EFFICIENT IO VIRTUALIZATION WITH IOMMU

WHAT ARE THE BENEFITS?

- Using the IOMMU allows a Hypervisor to assign a physical device exclusively to a Guest VM without danger of memory corruption to other VMs
  - Beneficial if one VM requires near native performance
  - Or if OS needs to be “sandboxed” (because of suspected malware)

- Native driver can operate in the Guest OS
EFFICIENT IO VIRTUALIZATION WITH IOMMU

WHAT ARE THE BENEFITS?

- Using the IOMMU allows a Hypervisor to assign a physical device exclusively to a Guest VM without danger of memory corruption to other VMs
  - Beneficial if one VM requires near native performance
  - Or if OS needs to be “sandboxed” (because of suspected malware)

- Native driver can operate in the Guest OS

- IOMMU enforces Hypervisor policy on memory and system resource isolation for each of the Guest Virtual Machines
EFFICIENT IO VIRTUALIZATION WITH IOMMU

WHAT ARE THE BENEFITS?

- Using the IOMMU allows a Hypervisor to assign a physical device exclusively to a Guest VM without danger of memory corruption to other VMs
  - Beneficial if one VM requires near native performance
  - Or if OS needs to be “sandboxed” (because of suspected malware)

- Native driver can operate in the Guest OS

- IOMMU enforces Hypervisor policy on memory and system resource isolation for each of the Guest Virtual Machines

- IOMMU redirects device physical address set up by Guest OS driver (= Guest Physical Addresses) to the actual Host System Physical Address (SPA)
EFFICIENT IO VIRTUALIZATION WITH IOMMU

WHAT ARE THE BENEFITS?

- Using the IOMMU allows a Hypervisor to assign a physical device exclusively to a Guest VM without danger of memory corruption to other VMs
  - Beneficial if one VM requires near native performance
  - Or if OS needs to be “sandboxed” (because of suspected malware)

- Native driver can operate in the Guest OS

- IOMMU enforces Hypervisor policy on memory and system resource isolation for each of the Guest Virtual Machines

- IOMMU redirects device physical address set up by Guest OS driver (= Guest Physical Addresses) to the actual Host System Physical Address (SPA)
  - Useful for platform resources that have “well-known” addresses like legacy devices or system resources like APIC (Advanced Programmable Interrupt Controller)
EFFICIENT IO VIRTUALIZATION WITH IOMMU

WHAT ARE THE BENEFITS?

- Using the IOMMU allows a Hypervisor to assign a physical device exclusively to a Guest VM without danger of memory corruption to other VMs
  – Beneficial if one VM requires near native performance
  – Or if OS needs to be “sandboxed” (because of suspected malware)

- Native driver can operate in the Guest OS

- IOMMU enforces Hypervisor policy on memory and system resource isolation for each of the Guest Virtual Machines

- IOMMU redirects device physical address set up by Guest OS driver (= Guest Physical Addresses) to the actual Host System Physical Address (SPA)
  – Useful for platform resources that have “well-known” addresses like legacy devices or system resources like APIC (Advanced Programmable Interrupt Controller)

- Allows near-native device performance for high-performance devices with low system impact
IOMMU USECASE: ENABLING HETEROGENEOUS COMPUTING
LEGACY GPU COMPUTE

The limiters that need to be fixed to unleash programmers:
The limiters that need to be fixed to unleash programmers:

- Multiple memory pools, multiple address spaces

**Diagram**

- **CPU** (Coherent)
- **GPU** (Non-Coherent)
- PCIe™

**LEGACY GPU COMPUTE**
The limiters that need to be fixed to unleash programmers:

- Multiple memory pools, multiple address spaces
- High overhead dispatch, data copies across PCIe
The limiters that need to be fixed to unleash programmers:

- Multiple memory pools, multiple address spaces
- High overhead dispatch, data copies across PCIe
- New languages and APIs for GPU programming necessary (OpenCL, etc.)
The limiters that need to be fixed to unleash programmers:

- Multiple memory pools, multiple address spaces
- High overhead dispatch, data copies across PCIe
- New languages and APIs for GPU programming necessary (OpenCL, etc.)
  - And sometimes proprietary environments
The limiters that need to be fixed to unleash programmers:

- Multiple memory pools, multiple address spaces
- High overhead dispatch, data copies across PCIe
- New languages and APIs for GPU programming necessary (OpenCL, etc.)
  - And sometimes proprietary environments

→ Dual source development
The limiters that need to be fixed to unleash programmers:

- Multiple memory pools, multiple address spaces
- High overhead dispatch, data copies across PCIe
- New languages and APIs for GPU programming necessary (OpenCL, etc.)
  - And sometimes proprietary environments
- Dual source development
- Expert programmers only
THE PREVIOUS APUS AND SOCS, PHYSICAL INTEGRATION

▲ Some memory copies are gone, because the same memory is accessed
Some memory copies are gone, because the same memory is accessed
– But the memory is not accessible concurrently, because of cache policies
Some memory copies are gone, because the same memory is accessed
– But the memory is not accessible concurrently, because of cache policies
Two memory pools remain (cache coherent + non-coherent memory regions)
Some memory copies are gone, because the same memory is accessed
– But the memory is not accessible concurrently, because of cache policies

Two memory pools remain (cache coherent + non-coherent memory regions)
Some memory copies are gone, because the same memory is accessed
  – But the memory is not accessible concurrently, because of cache policies
Two memory pools remain (cache coherent + non-coherent memory regions)
Jobs are still queued through the OS driver chain and suffer from overhead
Some memory copies are gone, because the same memory is accessed
- But the memory is not accessible concurrently, because of cache policies
Two memory pools remain (cache coherent + non-coherent memory regions)
Jobs are still queued through the OS driver chain and suffer from overhead
Still requires expert programmers to get performance
Some memory copies are gone, because the same memory is accessed
  – But the memory is not accessible concurrently, because of cache policies
Two memory pools remain (cache coherent + non-coherent memory regions)
Jobs are still queued through the OS driver chain and suffer from overhead
Still requires expert programmers to get performance
This is only an intermediate step in the journey
Unified Coherent Memory enables data sharing across all processors
AN HSA ENABLED SOC

- Unified Coherent Memory enables data sharing across all processors
AN HSA ENABLED SOC

- Unified Coherent Memory enables data sharing across all processors
- Processors architected to operate cooperatively
AN HSA ENABLED SOC

- Unified Coherent Memory enables data sharing across all processors
- Processors architected to operate cooperatively
  - Can exchange data “on the fly”, similar to what CPU threads do
AN HSA ENABLED SOC

- Unified Coherent Memory enables data sharing across all processors
- Processors architected to operate cooperatively
  - Can exchange data “on the fly”, similar to what CPU threads do
  - The lower job dispatch overhead allows tasks to be handled by the GPU that previously were “too costly” to transfer over
- Designed to enable the application running on different processors without substantially changing the programming logic
IOMMU: A BUILDING BLOCK FOR HSA
REDUCING THE OVERHEAD TO CALL THE GPU OR OTHER ACCELERATORS

The goals of the Heterogeneous System Architecture (HSA) and where the IOMMU helps:
The goals of the Heterogeneous System Architecture (HSA) and where the IOMMU helps:

- Use of accelerators as a first-class, peer processor within the system
The goals of the Heterogeneous System Architecture (HSA) and where the IOMMU helps:

- Use of accelerators as a first-class, peer processor within the system
  - **Unified process address space access across all processors**
    - **Shared Virtual Memory (SVM), “GPU ptr == CPU ptr”**
The goals of the Heterogeneous System Architecture (HSA) and where the IOMMU helps:

- Use of accelerators as a first-class, peer processor within the system
  - Unified process address space access across all processors
  - Shared Virtual Memory (SVM), “GPU ptr == CPU ptr”
IOMMU: A BUILDING BLOCK FOR HSA
REDUCING THE OVERHEAD TO CALL THE GPU OR OTHER ACCELERATORS

The goals of the Heterogeneous System Architecture (HSA) and where the IOMMU helps:

- Use of accelerators as a first-class, peer processor within the system
  - Unified process address space access across all processors
    - Shared Virtual Memory (SVM), “GPU ptr == CPU ptr”
  - Accelerator operates in **pageable system memory**

*Note: Pageable system memory refers to the ability of pages within the memory to be swapped between the CPU and the GPU, allowing efficient use of resources across different processors in the system.*
The goals of the Heterogeneous System Architecture (HSA) and where the IOMMU helps:

- Use of accelerators as a first-class, peer processor within the system
  - Unified process address space access across all processors
    - Shared Virtual Memory (SVM), “GPU ptr == CPU ptr”
  - Accelerator operates in **pageable system memory***

*with OS support & ATS/PRI
The goals of the Heterogeneous System Architecture (HSA) and where the IOMMU helps:

- Use of accelerators as a first-class, peer processor within the system
  - Unified process address space access across all processors
    - Shared Virtual Memory (SVM), “GPU ptr == CPU ptr”
  - Accelerator operates in **pageable system memory** *
  - Cache coherency between the CPU and accelerator caches
  - User mode dispatch/scheduling reduces job-dispatch overhead
  - QoS with preemption/context switch of GPU Compute Units

*with OS support & ATS/PRI
IOMMU: A BUILDING BLOCK FOR HSA
REDUCING THE OVERHEAD TO CALL THE GPU OR OTHER ACCELERATORS

The goals of the Heterogeneous System Architecture (HSA) and where the IOMMU helps:

- Use of accelerators as a first-class, peer processor within the system
  - Unified process address space access across all processors
    - Shared Virtual Memory (SVM), “GPU ptr == CPU ptr”
  - Accelerator operates in pageable system memory*
  - Cache coherency between the CPU and accelerator caches
  - User mode dispatch/scheduling reduces job-dispatch overhead
  - QoS with preemption/context switch of GPU Compute Units

- The IOMMU enforces control of GPU access to memory

*with OS support & ATS/PRI
IOMMU: A BUILDING BLOCK FOR HSA
REDUCING THE OVERHEAD TO CALL THE GPU OR OTHER ACCELERATORS

The goals of the Heterogeneous System Architecture (HSA) and where the IOMMU helps:

- Use of accelerators as a first-class, peer processor within the system
  - Unified process address space access across all processors
    - Shared Virtual Memory (SVM), “GPU ptr == CPU ptr”
  - Accelerator operates in **pageable system memory**
  - Cache coherency between the CPU and accelerator caches
  - User mode dispatch/scheduling reduces job-dispatch overhead
  - QoS with preemption/context switch of GPU Compute Units

- The IOMMU enforces control of GPU access to memory
  - OS can efficiently and safely share process page tables with accelerators (requires ATS/PRI protocol support)

*with OS support & ATS/PRI
The goals of the Heterogeneous System Architecture (HSA) and where the IOMMU helps:

- Use of accelerators as a first-class, peer processor within the system
  - Unified process address space access across all processors
    - Shared Virtual Memory (SVM), “GPU ptr == CPU ptr”
  - Accelerator operates in **pageable system memory***
  - Cache coherency between the CPU and accelerator caches
  - User mode dispatch/scheduling reduces job-dispatch overhead
  - QoS with preemption/context switch of GPU Compute Units

- The IOMMU enforces control of GPU access to memory
  - OS can efficiently and safely share process page tables with accelerators (requires ATS/PRI protocol support)
  - Accelerators can’t step outside of the OS-set boundaries

*with OS support & ATS/PRI
IOMMU: A BUILDING BLOCK FOR HSA
REDUCING THE OVERHEAD TO CALL THE GPU OR OTHER ACCELERATORS

The benefits of the Heterogeneous System Architecture:
IOMMU: A BUILDING BLOCK FOR HSA
REDUCING THE OVERHEAD TO CALL THE GPU OR OTHER ACCELERATORS

The benefits of the Heterogeneous System Architecture:

- Pageable memory access is validated and handled directly by the OS memory manager via AMD IOMMU
IOMMU: A BUILDING BLOCK FOR HSA
REDUCING THE OVERHEAD TO CALL THE GPU OR OTHER ACCELERATORS

The benefits of the Heterogeneous System Architecture:

- Pageable memory access is validated and handled directly by the OS memory manager via AMD IOMMU
- Application data structures can be directly parsed by the accelerator and pointer links followed without CPU help
IOMMU: A BUILDING BLOCK FOR HSA
REDUCING THE OVERHEAD TO CALL THE GPU OR OTHER ACCELERATORS

The benefits of the Heterogeneous System Architecture:

- Pageable memory access is validated and handled directly by the OS memory manager via AMD IOMMU
- Application data structures can be directly parsed by the accelerator and pointer links followed without CPU help
- Common high level languages and tools (compilers, runtimes, ...) port easily to accelerators
The benefits of the Heterogeneous System Architecture:

- Pageable memory access is validated and handled directly by the OS memory manager via AMD IOMMU
- Application data structures can be directly parsed by the accelerator and pointer links followed without CPU help
- Common high level languages and tools (compilers, runtimes, ...) port easily to accelerators
  - C/C++, Python, Java, ... already have open source implementations
IOMMU: A BUILDING BLOCK FOR HSA
REDUCING THE OVERHEAD TO CALL THE GPU OR OTHER ACCELERATORS

The benefits of the Heterogeneous System Architecture:

- Pageable memory access is validated and handled directly by the OS memory manager via AMD IOMMU
- Application data structures can be directly parsed by the accelerator and pointer links followed without CPU help
- Common high level languages and tools (compilers, runtimes, ...) port easily to accelerators
  - C/C++, Python, Java, ... already have open source implementations
  - Many more languages to follow
IOMMU: A BUILDING BLOCK FOR HSA
REDUCING THE OVERHEAD TO CALL THE GPU OR OTHER ACCELERATORS

The benefits of the Heterogeneous System Architecture:

- Pageable memory access is validated and handled directly by the OS memory manager via AMD IOMMU
- Application data structures can be directly parsed by the accelerator and pointer links followed without CPU help
- Common high level languages and tools (compilers, runtimes, ...) port easily to accelerators
  - C/C++, Python, Java, ... already have open source implementations
  - Many more languages to follow
- IOMMU making it easier for programmers to use GPUs and other accelerators safely and efficiently
Goal of the software stack is to focus on high-level language support

HSA Software Stack

Hardware - APUs, CPUs, GPUs

- User mode component
- Kernel mode component
- Components contributed by third parties

© Copyright 2014 HSA Foundation. All Rights Reserved.
Goal of the software stack is to focus on high-level language support.
Goal of the software stack is to focus on high-level language support
– Allow to target the GPU directly by SW
Goal of the software stack is to focus on high-level language support
- Allow to target the GPU directly by SW
- Drivers are setting up the HW and policies, then go out of the way
Goal of the software stack is to focus on high-level language support
- Allow to target the GPU directly by SW
- Drivers are setting up the HW and policies, then go out of the way
- IOMMU support provide hardware enforced protections for Operating System
LINES-OF-CODE AND PERFORMANCE COMPARISONS

(Exemplary ISV “Hessian” Kernel)

AMD A10-5800K APU with Radeon™ HD Graphics – CPU: 4 cores, 3800MHz (4200MHz Turbo); GPU: AMD Radeon HD 7660D, 6 compute units, 800MHz; 4GB RAM.
Software – Windows 7 Professional SP1 (64-bit OS); AMD OpenCL™ 1.2 AMD-APP (937.2); Microsoft Visual Studio 11 Beta

© Copyright 2014 HSA Foundation. All Rights Reserved.
LINES-OF-CODE AND PERFORMANCE COMPARISONS

AMD A10-5800K APU with Radeon™ HD Graphics – CPU: 4 cores, 3800MHz (4200MHz Turbo); GPU: AMD Radeon HD 7660D, 6 compute units, 800MHz; 4GB RAM. Software – Windows 7 Professional SP1 (64-bit OS); AMD OpenCL™ 1.2 AMD-APP (937.2); Microsoft Visual Studio 11 Beta

© Copyright 2014 HSA Foundation. All Rights Reserved.
ACCELERATORS: THE PORTABILITY CHALLENGE

CPU ISAs
- ISA innovations added incrementally (i.e., NEON, AVX, etc)
  - ISA retains backwards-compatibility with previous generation
- Two dominant instruction-set architectures: ARM and x86

GPU ISAs
- Massive diversity of architectures in the market
  - Each vendor has its own ISA - and often several in the market at same time
- No commitment (or attempt!) to provide any backwards compatibility
  - Traditionally graphics APIs (OpenGL, DirectX) provide necessary abstraction
WHAT IS HSA INTERMEDIATE LANGUAGE (HSAIL)?

Intermediate language for parallel compute in HSA
- Generated by a “High Level Compiler” (GCC, LLVM, Java VM, etc.)
- Expresses parallel regions of code
- Binary format of HSAIL is called “BRIG”
- Goal: Bring parallel acceleration to mainstream programming languages

IOMMU based pointer translation is key to enabling an efficient IL Implementation

```c
main() {
  ...
  #pragma omp parallel for
  for (int i=0; i<N; i++) {
    ...
  }
  ...
}
```

© Copyright 2014 HSA Foundation. All Rights Reserved.
FIR is a memory-intensive streaming workload

AES is a compute-intensive streaming workload

CL12 – cl_mem buffer
   - Copy to/from the device

CL20 – SVM buffer – Coarse Grain Sync
   - Copy to/from SVM
   - Data copy cannot be avoided, since the space for SVM is limited

HSA – Unified Memory Space – Fine Grained Sync
   - Regular pointer
   - No explicit copy

Results
   - HSA compute abstraction
   - NO performance penalty

Not all algorithms run faster
   - Measured on Kaveri (A pre-HSA 1.0 device)
   - Limited Coherent throughput

BLACKSCHOLES

- C++ on HSA
  - Matches or outperforms OpenCL
- Course Grained SVM
  - Matches OpenCL buffers for bandwidth
  - More predictable performance
- Fine Grained SVM
  - Faster kernel dispatch
  - Larger allocations
  - Shared data structure
- Results
  - HSA compute abstraction
  - NO performance penalty

SOURCE: RALPH POTTER – CODEPLAY. PRESENTATION MADE TO SG14 C++ WORKGROUP
ENABLING HETEROGENEOUS COMPUTING
SUMMARY AND DEMONSTRATION

Key Takeaways:

- To further scale up compute performance, software must take better advantage of system accelerators like GPUs and DSPs in high level languages.
- Accelerators following the HSA Foundation specification requirements allow programmers to write or port programs easily using common high level languages.
- AMD IOMMU is key to efficiently and safely access process virtual memory!
  - Does translation of both process address space via PASID and device physical accesses.
  - Enforces OS allocation policy, deals with virtual memory page faults, and much more.
AGENDA

INTERNALS

How does IOMMU work?

RESEARCH

Research Opportunities and Tools
RECAP: IOMMU AND ITS CAPABILITIES

Key capabilities:
1. Memory protection for DMA
2. Virtual address translation for DMA
3. Interrupt remapping and virtualization
4. IO can share CPU page tables
AGENDA: WHAT IS COMING UP?

---

DMA Address Translation
- Address translation and memory protection in un-virtualized System
- Making address translation faster through caching
- Enabling shared address space in heterogeneous system
- Enabling pre-translation through IOMMU
- Enabling demand paging from devices (dynamic page fault)
- Nested address translation in virtualized system
- Invalidating IOMMU mappings
AGENDA: WHAT IS COMING UP?

- DMA Address Translation
  - Address translation and memory protection in un-virtualized System
  - Making address translation faster through caching
  - Enabling shared address space in heterogeneous system
  - Enabling pre-translation through IOMMU
  - Enabling demand paging from devices (dynamic page fault)
  - Nested address translation in virtualized system
  - Invalidating IOMMU mappings

- Interrupt Handling
  - Interrupt filtering and remapping
  - Interrupt virtualization
AGENDA: WHAT IS COMING UP?

DMA Address Translation
- Address translation and memory protection in un-virtualized System
- Making address translation faster through caching
- Enabling shared address space in heterogeneous system
- Enabling pre-translation through IOMMU
- Enabling demand paging from devices (dynamic page fault)
- Nested address translation in virtualized system
- Invalidating IOMMU mappings

Interrupt Handling
- Interrupt filtering and remapping
- Interrupt virtualization

Summary
- A peek inside a typical IOMMU implementation
- Data structures and their Interactions

Address translation, memory protection, HSA

Interrupts
IOMMU Internals:
Address Translation and Memory Protection
ADDRESS TRANSLATION AND MEMORY PROTECTION
NON-VIRTUALIZED SYSTEM

Virtual Addresses

Core

Core

MMU

MMU

Physical Addresses

IO Device

IO Device

IOMMU

Memory
ADDRESS TRANSLATION AND MEMORY PROTECTION
NON-VIRTUALIZED SYSTEM

Virtual Addresses

Core

Core

MMU

MMU

Physical Addresses

IO Device

IO Device

IOMMU

Memory

Domain (Defined by OS)
ADDRESS TRANSLATION AND MEMORY PROTECTION
NON-VIRTUALIZED SYSTEM

Core

Virtual Addresses

MMU

Physical Addresses

Core

IO Device

Virtual Address

IO Device

DeviceID

Domain
(Defined by OS)

DMA Request

MMU

IOMMU

Device Table

Memory

DevID

DomID

IOMMU TUTORIAL @ ASPLOS | 3RD APRIL 2016
ADDRESS TRANSLATION AND MEMORY PROTECTION
NON-VIRTUALIZED SYSTEM

- Core
- Core
- MMU
- MMU
- IO Device
- IO Device
- IOMMU
- Memory
- Device Table
- Page Table
- Virtual Addresses
- Physical Addresses
- Domain (Defined by OS)

Virtual Address Request
DeviceID
Page Table

DevID
DomID

Device Table
ADDRESS TRANSLATION AND MEMORY PROTECTION
NON-VIRTUALIZED SYSTEM

Virtual Addresses
Core
MMU
Core

Physical Addresses

MMU

IO Device

Device ID

IOMMU

Domain (Defined by OS)

Virtual Address

 DMA Request

Physical Addresses

Memory

DevID

DomID

Device Table

Page Table

Physical Addresses

DevID

DomID

Device Table

IOMMU
ADDRESS TRANSLATION AND MEMORY PROTECTION
NON-VIRTUALIZED SYSTEM

Core  Core

Virtual Addresses

MMU  MMU

Physical Addresses

Abort request if not sufficient permission

IO Device  IO Device

IO Device

Virtual Address Request

DeviceID

DMA Request

IOMMU

Memory

DevID

Device Table

Page Table
MAKING TRANSLATION FAST
CACHING TRANSLATION IN IOMMU

Virtual Addresses

Core

MMU

MMU

Core

Physical Addresses

IO Device

Device Table

Entry Cache

Translation

Lookaside Buffer

Page Table Walker

Memory

Device Table

Page Table

DevID

Translation Lookaside Buffer

Page Table Walker
IOMMU Internals: Enabling “Pointer-is-a-Pointer” in Heterogeneous Systems
SHARING ADDRESS SPACE WITH CPU
ENABLING POINTER AS POINTER IN HETEROGENEOUS SYSTEMS
SHARING ADDRESS SPACE WITH CPU
ENABLING POINTER AS POINTER IN HETERogeneous SYSTEMS
SHARING ADDRESS SPACE WITH CPU
ENABLING POINTER AS POINTER IN HETEROGENEOUS SYSTEMS
SHARING ADDRESS SPACE WITH CPU
ENABLING POINTER AS POINTER IN HETEROGENEOUS SYSTEMS

Process

Core

Virtual Addresses

MMU

MMU

Physical Addresses

IO Device

GPU

Domain

Virtual Address Request

Physical Addresses

IOMMU

Device Table

Memory

x86-64 Page Table

DevID

190 IOMMU TUTORIAL @ ASPLOS | 3rd APRIL 2016
SHARING ADDRESS SPACE WITH CPU

ENABLING POINTER AS POINTER IN HETEROGENEOUS SYSTEMS

Virtual Addresses

MMU

Physical Addresses

MMU

Virtual Addresses

IOMMU

Virtual Address

Physica

Physical Addresses

x86-64 Page Table

Device Table

DevID

x86-64 Page Table
SHARING ADDRESS SPACE WITH CPU
ENABLING POINTER AS POINTER IN HETEROGENEOUS SYSTEMS

Needs ability to identify more than one address space
SHARING ADDRESS SPACE WITH CPU
ENABLING POINTER AS POINTER IN HETEROGENEOUS SYSTEMS

Virtual Addresses

MMU

Process 0

Virtual Address

MMU

Process 1

Physical Addresses

IO Device

GPU

Domain

Virtual Address Request

IOMMU

DeviceID

Memory

Device Table

Physical Addresses

Physical Addresses

DeviceID

Process 0

Process 1
SHARING ADDRESS SPACE WITH CPU
ENABLING POINTER AS POINTER IN HETEROGENEOUS SYSTEMS

Virtual Addresses

Physical Addresses

MMU

Process 0
PASID 0

Process 1
PASID 1

IO Device

GPU

Virtual Address Request

DMA Request

Physical Addresses

Memory

Device Table

DeviceID + PASID

Domain
SHARING ADDRESS SPACE WITH CPU
ENABLING POINTER AS POINTER IN HETEROGENEOUS SYSTEMS

Virtual Addresses

MMU

Physical Addresses

IO Device

Virtual Address Request

Device Table

PASID

DevID

Memory

Physical Addresses

GPU

DeviceID + PASID

Domain

IOMMU

DeviceID
SHARING ADDRESS SPACE WITH CPU
ENABLING POINTER AS POINTER IN HETEROGENEOUS SYSTEMS

Virtual Addresses

MMU

Physical Addresses

Memory

Device Table

DeviceID + PASID

Domain

Process 0
PASID 0

Process 1
PASID 1

Virtual Address

DMA Request

IO Device

GPU

Physical Addresses

IOMMU

Virtual Addresses

gCR3 table

DevID

PASID
SHARING ADDRESS SPACE WITH CPU
ENABLING POINTER AS POINTER IN HETEROGENEOUS SYSTEMS

Virtual Addresses

Physical Addresses

MMU

IO Device

GPU

Virtual Address Request

 DMA Request

Physical Addresses

Device Table

Memory

DevID

PASID

gCR3 table

DeviceID + PASID

Domain

Process 0
PASID 0

Process 1
PASID 1

Virtual Addresses

Physical Addresses

MMU

MMU
IOMMU Internals: Enabling Translation Caching in Devices
CACHING ADDRESS TRANSLATION IN DEVICES
ENABLING MORE CAPABLE DEVICE/ACCELERATORS

- Core
- Core
- MMU
- MMU
- Device Table Entry Cache
- Translation Lookaside Buffer
- Page Table Walker
- Memory
- Device Table
- gCR3 Table
- DevID
- PASID
Locally caching address translation in device reduces trips to IOMMU
CACHING ADDRESS TRANSFORMATION IN DEVICES
ENABLING MORE CAPABLE DEVICE/ACCELERATORS

IOMMU driver assigns per-translation capability to devices

ATC/ IOTLB

Device Table
Entry Cache
Translation
Lookaside Buffer
Page Table

Pre-translation capable?

DevID

Memory

IO Device

Core

Core

MMU

MMU
CACHING ADDRESS TRANSLATION IN DEVICES
ENABLING MORE CAPABLE DEVICE/ACCELERATORS

Introduce new message type:
Address Translation Service (ATS)

Device Table
Entry Cache
Translation Lookaside Buffer
Page Table

Memory

Device Table

IO Device

Core

Core

MMU

MMU

ATC/ IOTLB

DevID

PASID

gCR

Pre-translation capable:

Page Table walker

1
CACHING ADDRESS TRANSLATION IN DEVICES
ENABLING MORE CAPABLE DEVICE/ACCELERATORS
CACHING ADDRESS TRANSLATION IN DEVICES
ENABLING MORE CAPABLE DEVICE/ACCELERATORS
CACHING ADDRESS TRANSLATION IN DEVICES
ENABLING MORE CAPABLE DEVICE/ACCELERATORS

Core

Core

MMU

MMU

Memory

IO Device

Device Table

Entry Cache

DMA Req
(Physical Address)

ATC/ IOTLB

Pre-translated Req

Pre-translation capable?

Device Table

gCR3 table

DevID

PASID
CACHING ADDRESS TRANSLATION IN DEVICES
ENABLING MORE CAPABLE DEVICE/ACCELERATORS

- Core
- Core
- MMU
- MMU
- Memory
- IO Device
- ATC/ IOTLB
- Device Table
- Entry Cache
- DMA Req (Physical Address)
- Pre-translated Req
- Pre-translation capable?
- DevID
- PASID
- gCR3 table
- Page Table walker
- Device Table Entry Cache
- GPU TLB Core
- Core
CACHING ADDRESS TRANSLATION IN DEVICES
ENABLING MORE CAPABLE DEVICE/ACCELERATORS

- Core
- MMU
- Core
- MMU
- TLB
- IO Device
- ATC/ IOTLB
- DMA Req (Physical Address)
- Pre-translation capable?
- Abort if not pre-translation capable
- DevID
- Device Table
- gCR3 table
- Memory
IOMMU Internals:
Enabling Demand Paging from IO
→ No Need to Pin Memory
ENABLING DEMAND PAGING FROM DEVICE
SERVICING DEVICE PAGE FAULT

Device(s) access local TLB (ATC/IOTLB) first

Device Table
Entry Cache
Translation
Lookaside Buffer
Page Table walker

Core
Core

MMU
MMU

TLB
IO Device

Device Table
Entry Cache
Translation
Lookaside Buffer

Page Table walker

Device Table

DevID

1

PASID

gCR3 table

Device Table walker

GPU

Core

Core

GPU
ENABLING DEMAND PAGING FROM DEVICE
SERVICING DEVICE PAGE FAULT

On a (IO)TLB hit no access to IOMMU

Device Table
Entry Cache
Translation
Lookaside Buffer
Page Table walker

Device Table
walker

210 IOMMU TUTORIAL @ ASPLOS | 3RD APRIL 2016
ENABLING DEMAND PAGING FROM DEVICE
SERVICING DEVICE PAGE FAULT

Core

MMU

Core

MMU

TLB

IO Device

Device Table
Entry Cache
Translation
Lookaside Buffer
Page Table
walker

Device Table

walker

DevID

1

PASID

gCR3 table
ENABLING DEMAND PAGING FROM DEVICE
SERVICING DEVICE PAGE FAULT

Core → MMU → Core

IO Device

Device Table
Entry Cache
Translation Lookaside Buffer
Page Table walker

 ATS Req (DevID, PASID, VA, R/W)

TLB

(IO)TLB miss

Core (IO)TLB miss

Device Table

1

DevID

PASID

gCR3 table

Page Table walker

Device Table Entry Cache

Translation Lookaside Buffer
ENABLING DEMAND PAGING FROM DEVICE
SERVICING DEVICE PAGE FAULT

Core

Core

MMU

MMU

IO Device

Device Table
Entry Cache

Translation
Lookaside Buffer

Page Table walker

TLB

ATS Req
(DevID, PASID, VA, R/W)

(PO)TLB miss

GPU

Core

Core

Device Table

DevID

1

PASID

gCR3 table

 ATS Req (DevID, PASID, VA, R/W)

Page fault-
No valid PTE
ENABLING DEMAND PAGING FROM DEVICE
SERVICING DEVICE PAGE FAULT

Page fault - No valid PTE

Device Table
Entry Cache
Translation Lookaside Buffer
Page Table walker

Core
Core

MMU
MMU

TLB
IO Device

ATS Resp (NACK)

DevID
PASID

Device Table

gCR3 table
ENABLING DEMAND PAGING FROM DEVICE

SERVICING DEVICE PAGE FAULT

PPR* request
(DevID, PASID, VA, R/W)

Device Table
Entry Cache
Translation
Lookaside Buffer
Page Table walker

Device Table

1

PASID

DevID

gCR3 table

*PPR = Page Peripheral Request
ENABLING DEMAND PAGING FROM DEVICE
SERVICING DEVICE PAGE FAULT

Core → MMU → TLB → IO Device

PPR* request (DevID, PASID, VA, R/W)

Device Table
Entry Cache
Translation Lookaside Buffer
Page Table walker

PPR Log (circular buffer)

Device Table

gCR3 table

*PPR= Page Peripheral Request
ENABLING DEMAND PAGING FROM DEVICE
SERVICING DEVICE PAGE FAULT

Device Table
Entry Cache

Translation
Lookaside Buffer

Page Table walker

PPR Log (circular buffer)

Core

Core

MMU

MMU

TLB

IO Device

Device Table

Device Table

DevID

1

PASID

PASID
dID
Addr
Flag

gCR3 table

*PPR= Page Peripheral Request
ENABLING DEMAND PAGING FROM DEVICE
SERVICING DEVICE PAGE FAULT

Fault batching possible

PPR Log (circular buffer)

Device Table
Entry Cache
Translation Lookaside Buffer
Page Table walker

*PPR= Page Peripheral Request
ENABLING DEMAND PAGING FROM DEVICE
SERVICING DEVICE PAGE FAULT

Interrupt handler

Core

MMU

MMU

Interrupt

IO Device

Device Table
Entry Cache

Translation
Lookaside Buffer

Page Table walker

PPR Log
(circular buffer)

PASID dID Addr Flag

Device Table

DevID

1

PASID

gCR3 table

*PPR= Page Peripheral Request
ENABLING DEMAND PAGING FROM DEVICE
SERVICING DEVICE PAGE FAULT

Interrupt handler

Core
MMU

IO Device
Device Table
Entry Cache
Translation Lookaside Buffer
Page Table walker

TPR Log (circular buffer)
PASID dID Addr Flag

Device Table
DevID
1

gCR3 table

*PPR= Page Peripheral Request
ENABLING DEMAND PAGING FROM DEVICE
SERVICING DEVICE PAGE FAULT

Device Table
Entry Cache

Translation
Lookaside Buffer

Page Table
walker

Core

Interrupt handler

MMU

MMU

IO Device

Device Table

Entry Cache

Translation

Lookaside Buffer

Page Table
walker

Work Queue

PPR Log
(circular buffer)

PASID dID Addr Flag

DevID

Device Table

gCR3 table

*PPR= Page Peripheral Request
ENABLING DEMAND PAGING FROM DEVICE

SERVICING DEVICE PAGE FAULT

OS worker thread

MMU

Core

IO Device

Device Table
Entry Cache
Translation
Lookaside Buffer
Page Table walker

Work Queue
PPR Log
(circular buffer)

Device Table

gCR3 table

*PPR= Page Peripheral Request

PPR Log

Work Queue

Device Table

DevID

1

PASID

dID

Addr

Flag

OS worker thread

Core

IO Device

Device Table

Entry Cache

Translation

Lookaside Buffer

Page Table walker

Work Queue

PPR Log
(circular buffer)

Device Table

DevID

1

PASID

dID

Addr

Flag

*PPR= Page Peripheral Request
ENABLING DEMAND PAGING FROM DEVICE

SERVING DEVICE PAGE FAULT

OS worker thread

Service page fault

Fix the page table

Work Queue

PPR Log (circular buffer)

Device Table Entry Cache

Translation Lookaside Buffer

Page Table walker

IO Device

Device Table

gCR3 table

*PPR= Page Peripheral Request
ENABLING DEMAND PAGING FROM DEVICE

SERVICING DEVICE PAGE FAULT

OS worker thread

Service page fault

Fix the page table

Write PPR completion command

Command Buffer

Work Queue

PPR Log (circular buffer)

Device Table

MMU

Core

TLB

IO Device

Device Table Entry Cache

Translation Lookaside Buffer

Page Table walker

OS worker thread

Core

Fix the page table

Write PPR completion command

Command Buffer

Work Queue

PPR Log (circular buffer)

Device Table

MMU

Core

TLB

IO Device

Device Table Entry Cache

Translation Lookaside Buffer

Page Table walker

DevID

PPR Log

(circular buffer)

Work Queue

Device Table

gCR3 table

*PPR= Page Peripheral Request
ENABLING DEMAND PAGING FROM DEVICE

SERVICING DEVICE PAGE FAULT

Service page fault

OS worker thread

Fix the page table

Write PPR completion command

Work Queue
PPR Log (circular buffer)

Command Buffer

Device Table
Entry Cache
Translation Lookaside Buffer
Page Table walker

GPU TLB
Core

MMU

IO Device

Device Table

gCR3 table

*PPR= Page Peripheral Request

OS worker thread

Service page fault

Fix the page table

Write PPR completion command

Command Buffer

Device Table

gCR3 table

*PPR= Page Peripheral Request
ENABLING DEMAND PAGING FROM DEVICE

SERVICING DEVICE PAGE FAULT

OS worker thread

Service page fault

Fix the page table

Write PPR completion command

PPR response (DevID, PASID, VA,..)

Work Queue

PPR Log (circular buffer)

PASID dID Addr Flag

Device Table

MMU

Core

TLB

IO Device

Device Table Entry Cache

Translation Lookaside Buffer

Page Table walker

Command Buffer

*PPR= Page Peripheral Request

226 IOMMU TUTORIAL @ ASPLOS | 3rd APRIL 2016
ENABLING DEMAND PAGING FROM DEVICE

SERVICING DEVICE PAGE FAULT

Core

MMU

Core

MMU

IO Device

TLB

Device Table Entry Cache

Translation Lookaside Buffer

Page Table walker

Work Queue

PPR Log (circular buffer)

Command Buffer

Device Table

DevID

1

PASID

gCR3 table
ENABLING DEMAND PAGING FROM DEVICE
SERVICING DEVICE PAGE FAULT

Core ➔ MMU ➔ Device Table ➔ TLB ➔ IO Device ➔ MMU ➔ Core

Retry original request

ATS Req (DevID, PASID, VA, R/W)

Device Table Entry Cache
Translation Lookaside Buffer
Page Table walker

Command Buffer

Work Queue
PPR Log (circular buffer)

Core

MMU

TLB

IO Device

Core

MMU

Core

Translation Lookaside Buffer
Page Table walker

Command Buffer

Work Queue
PPR Log (circular buffer)

Device Table

gCR3 table
ENABLING DEMAND PAGING FROM DEVICE
SERVICING DEVICE PAGE FAULT

Core

Core

MMU

MMU

IO Device

Device Table
Entry Cache

Translation
Lookaside Buffer

Page Table walker

Retry original request

ATS Req
(DevID, PASID, VA, R/W)

Command Buffer

Device Table

PPR Log (circular buffer)

Work Queue

DevID

1

PASID

gCR3 table

Core

Core

TLB

IO Device

Device Table Entry Cache

Translation Lookaside Buffer

Page Table walker

Command Buffer

Device Table

PPR Log (circular buffer)

Work Queue

DevID

1

PASID

gCR3 table

Core

Core

MMU

MMU

IO Device

Device Table
Entry Cache

Translation
Lookaside Buffer

Page Table walker

Retry original request

ATS Req
(DevID, PASID, VA, R/W)
ENABLING DEMAND PAGING FROM DEVICE
SERVICING DEVICE PAGE FAULT

Core → MMU

Core → MMU

Thread

Device Table

Entry Cache

Translation Lookaside Buffer

Page Table walker

 ATS Resp (PASID, VA, PA, Attr.)

MMU

IO Device

TLB

Command Buffer

Work Queue

PPR Log (circular buffer)

DevID

Device Table

PASID

gCR3 table

Retry original request
ENABLING DEMAND PAGING FROM DEVICE
SERVICING DEVICE PAGE FAULT

Core -> MMU -> TLB -> IO Device

DMA Req (Physical Address)

Retry original request

Device Table
Entry Cache
Translation Lookaside Buffer
Page Table walker

Command Buffer

Work Queue
PPR Log (circular buffer)

Device Table

gCR3 table
IOMMU Internals:
Nested (Two-Level) Address Translation
RECAP: ADDRESS TRANSLATION IN VIRTUALIZED SYSTEMS

Guest Virtual Address (GVA)

Guest Page Table (GPT)

Guest Physical Address (GPA)

Host Page Table (HPT)

System Physical Address (SPA)

Hypervisor (a.k.a. VMM)

Guest Applications

Guest OS 0

Guest OS 1

Hardware – CPU, Memory, IO

Guest OS does not have access to (system) physical address
NESTED ADDRESS TRANSLATION BY IOMMU

Guest OS 0

Guest OS 1

VMM

Io Device

GPU

IOMMU

Memory

Device Table

DevID

DevID

GVA

GPT

GPA

HPT

SPA

Guest Process

Guest Process

Core 0

Core 0

Domain

MMU

MMU
NESTED ADDRESS TRANSLATION BY IOMMU

Guest OS 0

Guest OS 1

VMM

Core 0

Core 0

MMU

MMU

IOMMU

IO Device

GPU

Domain

Device ID + PASID

Guest Virtual Address

DMA Request

IOMMU

Memory

Device Table

GVA

GPT

GPA

HPT

SPA
NESTED ADDRESS TRANSLATION BY IOMMU

Guest OS 0
Guest OS 1

Core 0
Core 0

VMM

IO Device

GPU

Device ID + PASID

Guest Virtual Address

DMA Request

Physical Addresses

Host Page Table

Guest Page Table(s)

Device Table

PASID

gCR3 table

DevID

Host Page Table

Guest Page Table(s)

Memory

GVA

GPT

GPA

HPT

SPA

GVA

GPT

GPA

HPT

SPA

IOMMU

MMU

MMU

Device Table

IOMMU

MMU

MMU

IOMMU

MMU

MMU

Domain

Guest Process

Identified by PASID

Identified by DevID/DomID

Guest Process

Identified by PASID

Identified by DevID/DomID

Guest Process
NESTED ADDRESS TRANSLATION BY IOMMU

GVA

Device Table Entry

HPT

Nested/Host page table

SPA

GVA [47:39]

GVA [38:30]

GVA [29:21]

GVA [20:12]

GVA [11:0]
IOMMU Internals:
Sending Commands to IOMMU
COMMANDS TO IOMMU

- IOMMU Driver (running on CPU) issues commands to IOMMU
  - e.g., Invalidate IOMMU TLB Entry, Invalidate IOTLB Entry
  - e.g., Invalidate Device Table Entry
  - e.g., Complete PPR, Completion Wait, etc.

- Issued via **Command Buffer**
  - Memory resident circular buffer
  - MMIO registers: Base, Head, and Tail register
IOMMU Driver (running on CPU) issues commands to IOMMU
- e.g., Invalidate IOMMU TLB Entry, Invalidate IOTLB Entry
- e.g., Invalidate Device Table Entry
- e.g., Complete PPR, Completion Wait, etc.

Issued via **Command Buffer**
- Memory resident circular buffer
- MMIO registers: Base, Head, and Tail register
EXAMPLE: IOMMU TLB SHOOTDOWN

- IOMMU TLB Shootdown
  - Update page table information
  - Flush TLB Entry(s) containing stale information

- Three steps in IOMMU TLB shootdown
  - Invalidating IOMMU TLB entry
  - Invalidating IO TLB (Device TLB) entry
  - Wait for completion
EXAMPLE: IOMMU TLB SHOOTDOWN

- IOMMU Driver
- Core
- MMU
- Command Buffer
- TLB
- IO Device
- Device Table
- Entry Cache
- Translation Lookaside Buffer
- Page Table Walker

IOMMU Driver

Core

MMU

Command Buffer

MMU

TLB

IO Device

Device Table

Entry Cache

Translation Lookaside Buffer

Page Table Walker
EXAMPLE: IOMMU TLB SHOOTDOWN

IOMMU Driver

Core

Core

MMU

MMU

Command Buffer

TLB

IO Device

Device Table
Entry Cache

Translation
Lookaside Buffer

Page Table walker


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry

IOMMU Driver

IO Device

Command Buffer

TLB


128 bits

invalide iommu tlb entry
EXAMPLE: IOMMU TLB SHOOTDOWN

IOMMU Driver

Core

Core

MMU

MMU

Command Buffer

TLB

IO Device

Device Table
Entry Cache

Translation
Lookaside Buffer

Page Table
walker
EXAMPLE: IOMMU TLB SHOOTDOWN

IOMMU Driver

Core

Core

MMU

MMU

IO Device

Device Table

Entry Cache

Translation

Lookaside Buffer

Page Table

walker

Command Buffer


128 bits

invalidate IO tlb entry
EXAMPLE: IOMMU TLB SHOOTDOWN

IOMMU Driver

Core

Core

MMU

MMU

IOMMU TLB SHOOTDOWN

IO Device

Device Table Entry Cache

Translation Lookaside Buffer

Page Table walker

Command Buffer

MMU Page Table walker
EXAMPLE: IOMMU TLB SHOOTDOWN

IOMMU Driver

Core

MMU

Core

MMU

IO Device

TLB

Device Table
Entry Cache

Translation
Lookaside Buffer

Page Table
cacher

c

Command Buffer

OpCode
Store Address
Store Data

128 bits

completion wait
EXAMPLE: IOMMU TLB SHOOTDOWN

IOMMU Driver

Core

Core

MMU

MMU

IO Device

TLB

Device Table Entry Cache

Translation Lookaside Buffer

Page Table walker

Command Buffer
EXAMPLE: IOMMU TLB SHOOTDOWN

- MMU
- Core
- Command Buffer
- Update Tail pointer
- Translation Lookaside Buffer
- Page Table walker
- Device Table Entry Cache
- IO Device
- TLB
- IOMMU Driver
- Entry Cache
EXAMPLE: IOMMU TLB SHOOTDOWN

- Core
- Core
- MMU
- MMU
- IOMMU Driver

- IO Device
- TLB
- Entry Cache
- Translation Lookaside Buffer
- Page Table Walker

- Command Buffer
EXAMPLE: IOMMU TLB SHOOTDOWN

IOMMU Driver

Core

MMU

Core

MMU

TLB

IO Device

Device Table
Entry Cache

Translation
Lookaside Buffer

Page Table
Walker

Command Buffer


invalidate IOMMU tlb entry

128 bits
EXAMPLE: IOMMU TLB SHOOTDOWN

IOMMU Driver

Core -> MMU -> Core

IO Device

Device Table Entry Cache
Translation Lookaside Buffer
Page Table Walker

Command Buffer

invalidate IOMMU tlb entry
128 bits
EXAMPLE: IOMMU TLB SHOOTDOWN

IOMMU Driver

Core

MMU

Core

MMU

Update Head pointer

Command Buffer

IOMMU TLB SHOOTDOWN

IO Device

Device Table
Entry Cache

Translation
Lookaside Buffer

Page Table
Walker
EXAMPLE: IOMMU TLB SHOOTDOWN

IOMMU Driver

Core

Core

MMU

MMU

TLB

IO Device

Device Table
Entry Cache

Translation
Lookaside Buffer

Page Table
Walker

Command Buffer


128 bits

invalidate IO tlb entry
EXAMPLE: IOMMU TLB SHOOTDOWN

IOMMU Driver

Core  Core

MMU  MMU

Command Buffer

Update Head pointer

IO Device

TLB

Device Table Entry Cache
Translation Lookaside Buffer
Page Table Walker

GPU

Core

Core
EXAMPLE: IOMMU TLB SHOOTDOWN

IOMMU Driver

Core

Core

MMU

MMU

Command Buffer

OpCode

Store Address

Store Data

128 bits

completion wait
EXAMPLE: IOMMU TLB SHOOTDOWN
EXAMPLE: IOMMU TLB SHOOTDOWN

- IOMMU Driver
- Core
- Core
- MMU
- MMU
- IO Device
- TLB
- Device Table
- Entry Cache
- Translation Lookaside Buffer
- Page Table Walker
- Command Buffer
- Wait for previous commands to finish

ACK
EXAMPLE: IOMMU TLB SHOOTDOWN

IOMMU Driver

Core

Core

MMU

MMU

IO Device

TLB

Device Table Entry Cache

Translation Lookaside Buffer

Page Table Walker

Command Buffer

Wait for previous commands to finish

IOMMU Stores Data to “Store Address” Or Raise Interrupt
EXAMPLE: IOMMU TLB SHOOTDOWN

IOMMU Driver

Core

Core

MMU

MMU

TLB

IO Device

Device Table
Entry Cache

Translation
Lookaside Buffer

Page Table
Walker

Update Head pointer

Command Buffer

Wait for previous commands to finish

Command Buffer

MMU

Translation Lookaside Buffer
IOMMU INTERNALS: INTERRUPT REMAPPING AND VIRTUALIZATION
INTERRUPT REMAPPING

Diagram showing the relationship between cores, MMUs, IO devices, and memory, with emphasis on interrupt remapping and the use of device tables, entry caches, lookaside buffers, and table walkers.
INTERRUPT REMAPPING

Core APIC → MMU

Core APIC → MMU

Memory

Fixed/Arbitrated Interrupt

Fixed/Arbitrated Interrupt

IO Device

Device Table Entry Cache

Interrupt Remapping Lookaside Buffer

Table Walker

Interrupt Remapping Lookaside Buffer

Table Walker

Device Table Entry Cache

Interrupt Remapping Lookaside Buffer

Table Walker
INTERRUPT REMAPPING

- Core
  - APIC
  - MMU

- IO Device
  - Entry Cache
  - Interrupt Remapping
  - Lookaside Buffer
  - Table walker

- Fixed/Arbitrated Interrupt

- Device Table

- Memory
  - DevID
  - Did
  - Device Table
INTERRUPT REMAPPING

- Core
- APIC
- MMU
- IO Device
- Entry Cache
- Interrupt Remapping Lookaside Buffer
- Table walker
- Memory
- DevID
- Device Table
INTERRUPT REMAPPING

Abort request if not sufficient permission
INTERRUPT REMAPPING

Core

APIC

MMU

Core

APIC

MMU

IO Device

IO Device

Device Table Entry Cache

Interrupt Remapping Lookaside Buffer

Table walker

Fixed/Arbitrated Interrupt

Memory
INTERRUPT VIRTUALIZATION

- Core
  - APIC
- VMM
- MMU
- Memory
- Guest OS 0
  - vAPIC
- IO Device
- IO Device
INTERRUPT VIRTUALIZATION

![Diagram of interrupt virtualization process]

- VMM
- Guest OS 0
- vAPIC
- Core APIC
- Core APIC
- MMU
- MMU
- IO Device
- IO Device
- Guest Virtualized Interrupt
- Memory
- Device Table
- DevID
- Did
- Interrupt Remapping Table
- Guest Mode
INTERRUPT VIRTUALIZATION

Abort request if not sufficient permission
INTERRUPT VIRTUALIZATION

Guest OS 0
vAPIC

Core APIC

Core APIC

VMM

MMU

MMU

IO Device

IO Device

Guest Virtualized Interrupt

Memory
Guest vAPIC backing page
Interrupt Virtualization

- Guest OS
- MMU
- Core
- APIC
- IO Device
- Memory
- VMM

DevID

Device Table

Did
INTERRUPT VIRTUALIZATION

- Guest OS 0
- vAPIC
- Core
- APIC
- VMM
- MMU
- IO Device
- Guest Virtualized Interrupt
- Memory
- Device Table
- Did
- DevID
- Interrrupt Remapping Table
- Guest Running
INTERRUPT VIRTUALIZATION

Guest OS 0

Inactive Guest

Core APIC
Core APIC

VMM

MMU

MMU

IO Device

IO Device

Guest Virtualized Interrupt

Memory
INTERRUPT VIRTUALIZATION

Guest OS 0
vAPIC

Inactive Guest

Core
APIC

Core
APIC

VMM

MMU

MMU

IO Device

IO Device

Guest Virtualized Interrupt

Memory

Device Table

Did

DevID

Guest NOT Running

Interrupt Remapping Table
INTERRUPT VIRTUALIZATION

Guest OS 0

vAPIC

Inactive Guest

Core

Core

VMM

APIC

APIC

MMU

MMU

IO Device

IO Device

Guest Virtualized Interrupt

Memory

Guest vAPIC Log

Guest

Virtualized

Interrupt

Log

Guest OS 0

vAPIC

Inactive

Guest
INTERRUPT VIRTUALIZATION

- Guest OS
- vAPIC
- Inactive Guest
- APIC
- Guest Virtualized Interrupt
- IO Device
- MMU
- Memory
- Guest vAPIC Log
INTERRUPT VIRTUALIZATION

Guest OS 0

vAPIC

Activate Target
Guest

Core

APIC

VMM

Guest Virtualized Interrupt

MMU

IO Device

IO Device

Memory
Interrupt Virtualization

Guest OS 0

vAPIC

Core

APIC

VMM

MMU

Guest Virtualized Interrupt

Memory

IO Device

IO Device
IOMMU internals: a typical IOMMU hardware design
EXAMPLE OF IOMMU HARDWARE DESIGN

DRAM

CPU
Memory Controller

IOHUB

IOMMU

Table Walker
L2 DTC
L2 ITC
L2 gPDC
L2 gPTC
L2 nPDC
L2 nPTC

Device

Device

Device
CACHE SIZING VS PRODUCT TYPE

Typical Client Product
- Non-Virtualized
- I/O Isolation
- Small Working Set
CACHE SIZING VS PRODUCT TYPE

Typical Server Product
- Virtualized
- Large Working Set
IOMMU INTERNALS: SUMMARY OF KEY DATA STRUCTURES
IOMMU’S KEY DATA STRUCTURES

IOMMU

- Device Table Base Register
- Command Buffer Base Register
- Event Log Base Register
- Page Request Log Base Register

DRAM

- Device Table
- GCR3 Table
- Guest Page Tables
- Host Page Tables
- Guest Virtual APIC Backing Page

Guest
- Guest vAPIC Log Base Register
- Command Buffer
- Event Log
- Peripheral Page Request Log

Interrupt Remap Table
DEVICE TABLE ENTRY

Each entry is 32B

- IOTLB Enable
- Interrupt info
  - Interrupt Table Root Pointer
  - Legacy Interrupt Permission
- guest translation Info
  - GCR3 Table Root Pointer
  - Guest Levels translated
- host translation Info
  - Page Mode
  - Host Page Table Root Pointer
- domainID
- valid entry

Notes:
- Each device table entry is 32B in size.
- The entry includes information for both host and guest translation, as well as interrupt handling.
- The diagram illustrates the layout of the entry with specific fields dedicated to different functionalities.
INTERRUPT REMAPPING TABLE ENTRY

Each entry is 128b. Two modes:

- Interrupt Remapping (guest mode=0)
- Interrupt Virtualization (guest mode=1)

**guest mode=0:**

**guest mode=1:**

- Guest vAPIC info
  - Guest vAPIC Root Pointer
  - Guest vAPIC Tag
  - Guest Running
AGENDA

Research Opportunities and Tools
Isolation from malicious or buggy third party accelerators

– Can IOMMU ensure protection in-presentation of untrusted accelerators?
RESEARCH DIRECTIONS

中国大陆 Isolation from malicious or buggy third party accelerators
  – Can IOMMU ensure protection in-prence of untrusted accelerators?

中国大陆 Specializing IOMMU for performance and power
  – Can IOMMU hardware exploit predictable access pattern of some accelerators?
RESEARCH DIRECTIONS

- Isolation from malicious or buggy third party accelerators
  - Can IOMMU ensure protection in presence of untrusted accelerators?
- Specializing IOMMU for performance and power
  - Can IOMMU hardware exploit predictable access pattern of some accelerators?
- Trading memory protection for performance
RESEARCH DIRECTIONS

- Isolation from malicious or buggy third party accelerators
  - Can IOMMU ensure protection in-presence of untrusted accelerators?

- Specializing IOMMU for performance and power
  - Can IOMMU hardware exploit predictable access pattern of some accelerators?

- Trading memory protection for performance
  - Can selectively lowering protection enable better performance?
RESEARCH DIRECTIONS

- Isolation from malicious or buggy third party accelerators
  - Can IOMMU ensure protection in-presence of untrusted accelerators?

- Specializing IOMMU for performance and power
  - Can IOMMU hardware exploit predictable access pattern of some accelerators?

- Trading memory protection for performance
  - Can selectively lowering protection enable better performance?

- Extending (limited) virtual memory to embedded accelerators
  - Can we design for IOMMU^LITE embedded low-power accelerators?
RESEARCH DIRECTIONS

- Isolation from malicious or buggy third party accelerators
  - Can IOMMU ensure protection in-preservation of untrusted accelerators?

- Specializing IOMMU for performance and power
  - Can IOMMU hardware exploit predictable access pattern of some accelerators?

- Trading memory protection for performance
  - Can selectively lowering protection enable better performance?

- Extending (limited) virtual memory to embedded accelerators
  - Can we design for IOMMU LITE embedded low-power accelerators?

- Avoiding interference in the IOMMU
  - How to reduce interference among multiple devices accessing IOMMU?
ISOLATION FROM THIRD PARTY ACCELERATORS
EMERGENCE OF 3rd PARTY ACCELERATORS

1st Party (Trusted)

Core

Core

MMU

MMU

Accelerator

Accelerator

IOMMU

Memory
ISOLATION FROM THIRD PARTY ACCELERATORS
EMERGENCE OF 3RD PARTY ACCELERATORS

Core

Core

MMU

MMU

Accelerator

Accelerator

IOMMU

3rd Party (Un-trusted)

Memory

Core

IOMMU TUTORIAL @ ASPLOS | 3RD APRIL 2016
Q: How to integrate third party accelerators efficiently and securely?

How to determine if a device is trustworthy and remains trustworthy?

May not be possible verify if 3rd party accelerator is not buggy.
ISOLATION FROM THIRD PARTY ACCELERATORS (CNTD.)

EMERGENCE OF 3\textsuperscript{RD} PARTY ACCELERATORS

3rd Party
(Un-trusted)
ISOLATION FROM THIRD PARTY ACCELERATORS (CNTD.)

EMERGENCE OF 3\textsuperscript{RD} PARTY ACCELERATORS

3rd Party
(Un-trusted)

Core

Core

Physical address

MMU

MMU

TLB

Accelerator

IOMMU

Performance consideration:
1. TLBs in accelerator \(\rightarrow\) Possible to bypass IOMMU
ISOLATION FROM THIRD PARTY ACCELERATORS (CNTD.)
EMERGENCE OF 3\textsuperscript{RD} PARTY ACCELERATORS

Performance consideration:
1. TLBs in accelerator $\rightarrow$ Possible to bypass IOMMU
2. Coherent caches in accelerator $\rightarrow$ Coherence traffic bypass IOMMU
ISOLATION FROM THIRD PARTY ACCELERATORS (CNTD.)

EMERGENCE OF 3rd PARTY ACCELERATORS

Related work:
Olson et al. “Border Control” in MICRO’15 [OLSON’15]
ISOLATION FROM THIRD PARTY ACCELERATORS (CNTD.)

EMERGENCE OF 3RD PARTY ACCELERATORS

Related work:
Olson et al. “Border Control” in MICRO’15 [OLSON’15]

Idea: Check every access with physical address if valid.
SPECIALIZING IOMMU FOR DEVICE/ACCELERATOR

- IOMMU design(s) resembles CPU MMU design
  - But device/accelerator access patterns differs from CPU’s

- IOMMU caters to disparate devices
  - Single design point may not be optimal for all
    - e.g., access pattern from GPU likely different from NIC’s
SPECIALIZING IOMMU FOR DEVICE/ACCELERATOR

▲ IOMMU design(s) resembles CPU MMU design
  – But device/accelerator access patterns differs from CPU’s

▲ IOMMU caters to disparate devices
  – Single design point may not be optimal for all
  – e.g., access pattern from GPU likely different from NIC’s

Study traffic pattern to IOMMU and specialize for common patterns

▲ Related work: Malka et al.’s “rIOMMU” in ASPLOS’15.
  – Idea: Exploit predictable IOMMU accesses from devices using circular ring buffers
SPECIALIZING IOMMU FOR DEVICE/ACCELERATOR

- IOMMU design(s) resembles CPU MMU design
  - But device/accelerator access patterns differs from CPU’s
- IOMMU caters to disparate devices
  - Single design point may not be optimal for all
  - e.g., access pattern from GPU likely different from NIC’s

**Study traffic pattern to IOMMU and specialize for common patterns**

- **Related work:** Malka et al.’s “rIOMMU” in ASPLOS’15.
  - **Idea:** Exploit **predictable** IOMMU accesses from devices using circular ring buffers
  - Replace page table with circular, flat table → Easy page walk
  - Predictable access → single entry IOTLB with no TLB miss and less invalidation
IOMMU design(s) resembles CPU MMU design
  - But device/accelerator access patterns differs from CPU’s

IOMMU caters to disparate devices
  - Single design point may not be optimal for all
  - e.g., access pattern from GPU likely different from NIC’s

Study traffic pattern to IOMMU and specialize for common patterns

Related work: Malka et al. ’s “rIOMMU” in ASPLOS’15.
  - Idea: Exploit predictable IOMMU accesses from devices using circular ring buffers
  - Replace page table with circular, flat table $\rightarrow$ Easy page walk
  - Predictable access $\rightarrow$ single entry IOTLB with no TLB miss and less invalidation

Possible to use device-specific knowledge to optimize performance
  - IOMMU prefetching and TLB caching hints can be useful
  - Replacement policy coordination between IOTLB (Device TLB) and IOMMU TLB
  - Energy/power optimization in IOMMU
IOMMU hardware allows lowering protection for performance

- For example: pre-translated DMA transactions pass-through IOMMU
- A trusted IO device can manipulate any address, including interrupt storms
IOMMU hardware allows lowering protection for performance
- For example: pre-translated DMA transactions pass-through IOMMU
- A trusted IO device can manipulate any address, including interrupt storms

OS policies for trading off protection for security
- Should the sysadmin decide how much to trust a device/driver?
- Exposing software knobs for dialing performance vs. protection
- Related work: OS policies for Strict vs Deferred protection strategy
  [WILMANN’08, BEN-YEHUDA’07, AMIT’11]
- ASPLOS’16: Strict, sub-page grain protection through Shadow DMA-buffer
  [MARKUZE’16]
Virtual memory eases programming (e.g., “pointer-is-pointer”)  
- But comes at performance and energy cost

Stripped-down IOMMU for ultra low-power accelerators  
- Lower hardware, performance, power cost by stripping non-essential features  
- Example “non-essential” features: IO virtualization support, Interrupt remapping, Page fault handling, Nested page table walker, etc.
Virtual memory eases programming (e.g., “pointer-is-pointer”)  
- But comes at performance and energy cost

Stripped-down IOMMU for **ultra low-power** accelerators  
- Lower hardware, performance, power cost by stripping non-essential features  
- Example “non-essential” features: IO virtualization support, Interrupt remapping, Page fault handling, Nested page table walker, etc.

**Related work:**
- Vogel et al.’s “Lightweight Virtual Memory” in CODES’15 [VOGEL’15]  
  - Idea: Software managed IOMMU for FPGA → No translation miss handling in hardware  
  - Simple design, high performance with effective software management
AVOIDING (DESTRUCTIVE-) INTERFERENCE IN IOMMU

Virtual Addresses

Core

MMU

Core

MMU

Physical Addresses

Memory

IO Device

IO Device

IOMMU
AVOIDING (DESTRUCTIVE-) INTERFERENCE IN IOMMU

Virtual Addresses

Core

Core

MMU

MMU

Physical Addresses

GPU

NIC

IOMMU

Memory
AVOIDING (DESTRUCTIVE-) INTERFERENCE IN IOMMU

Diagram showing the relationship between Virtual Addresses, Physical Addresses, MMU, IOMMU, GPU, NIC, and Memory.
AVOIDING (DESTRUCTIVE-) INTERFERENCE IN IOMMU

IOMMU is a shared resource

How to model contention in IOMMU?
How to guarantee Quality-of-Service in IOMMU?
Software research: IOMMU driver/OS policies
- Easy! Open source IOMMU Driver in Linux

Hardware research: Modifying IOMMU hardware behavior
- Option 1: Hardware performance counter + Analytical models
- Option 2: Simulator with IOMMU model
  - Work in progress to add IOMMU model in gem5
  - Write down in attendance sheet your email if interested
IOMMU (kernel-mode) Driver:
Configuration/Setup IOMMU hardware

Important roles:
1. Memory protection from rogue devices
2. Shared virtual memory to devices
3. I/O virtualization – direct I/O
4. Supporting legacy I/O, Secure boot
REFERENCES

- IOMMU specification: [http://support.amd.com/TechDocs/48882_IOMMU.pdf](http://support.amd.com/TechDocs/48882_IOMMU.pdf)
- VOGEL’15: Pirmin Vogel et. al. “Lightweight virtual memory support for many-core accelerators in heterogeneous embedded SoCs”, CODES’15
QUESTIONS AND FEEDBACK

Reachable @

- Arka Basu: Arkaprava “dot” Basu “at” amd.com
- Andy Kegel: Andrew “dot” Kegel “at” amd.com
- Paul Blinzer: Paul “dot” Blinzer “at” amd.com
- Maggie Chan: Maggie “dot” Chan “at” amd.com
DISCLAIMER & ATTRIBUTION

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION

© 2016 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. SPEC is a registered trademark of the Standard Performance Evaluation Corporation (SPEC). OpenCL is a trademark of Apple Inc. used by permission by Khronos. ARM® is/are the registered trademark(s) of ARM Limited in the EU and other countries. PCIe® is registered trademark of PCI-SIG corporation. Other name are for informational purposes only and may be trademarks of their respective owners.