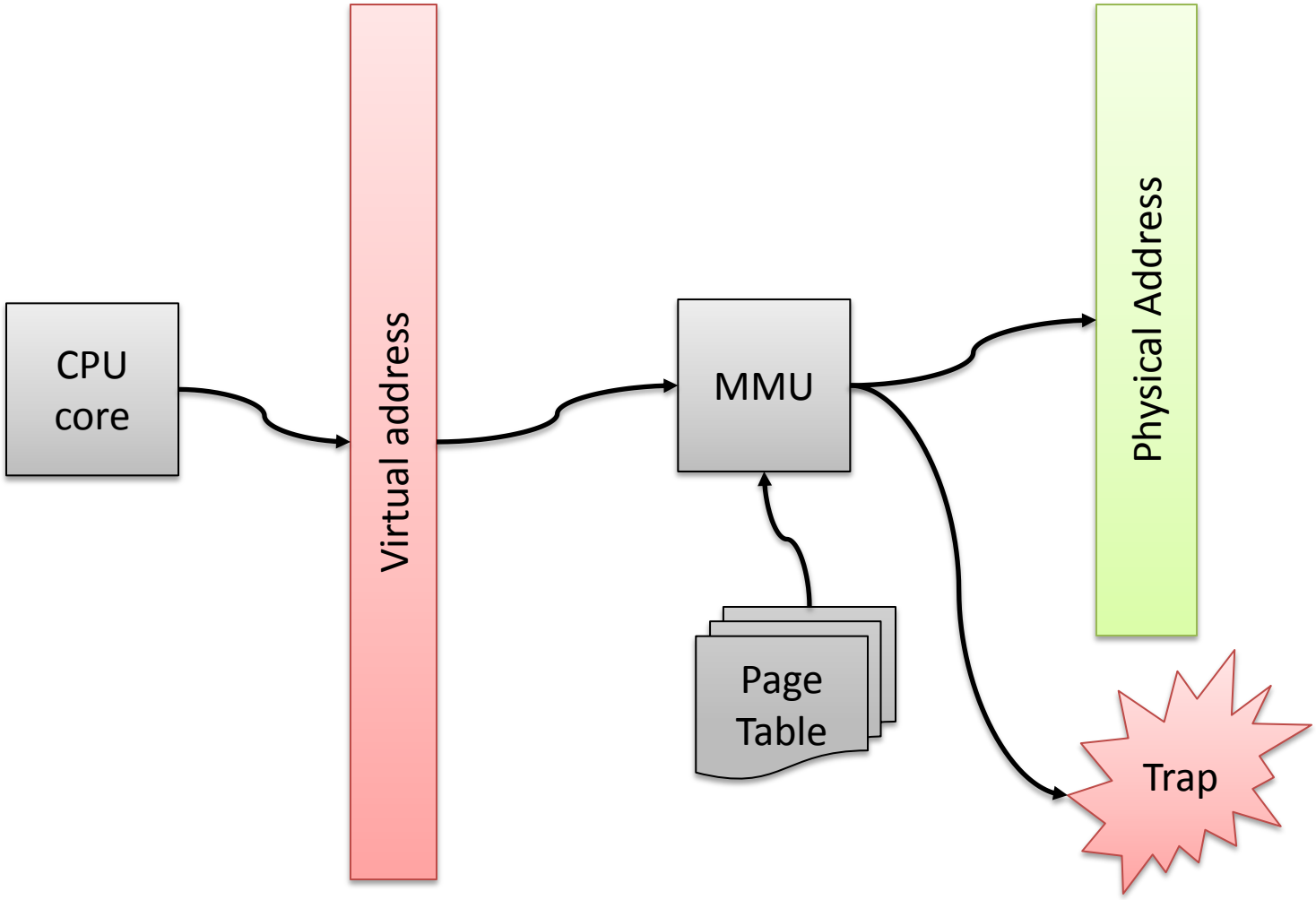
Self-Paging

This week:

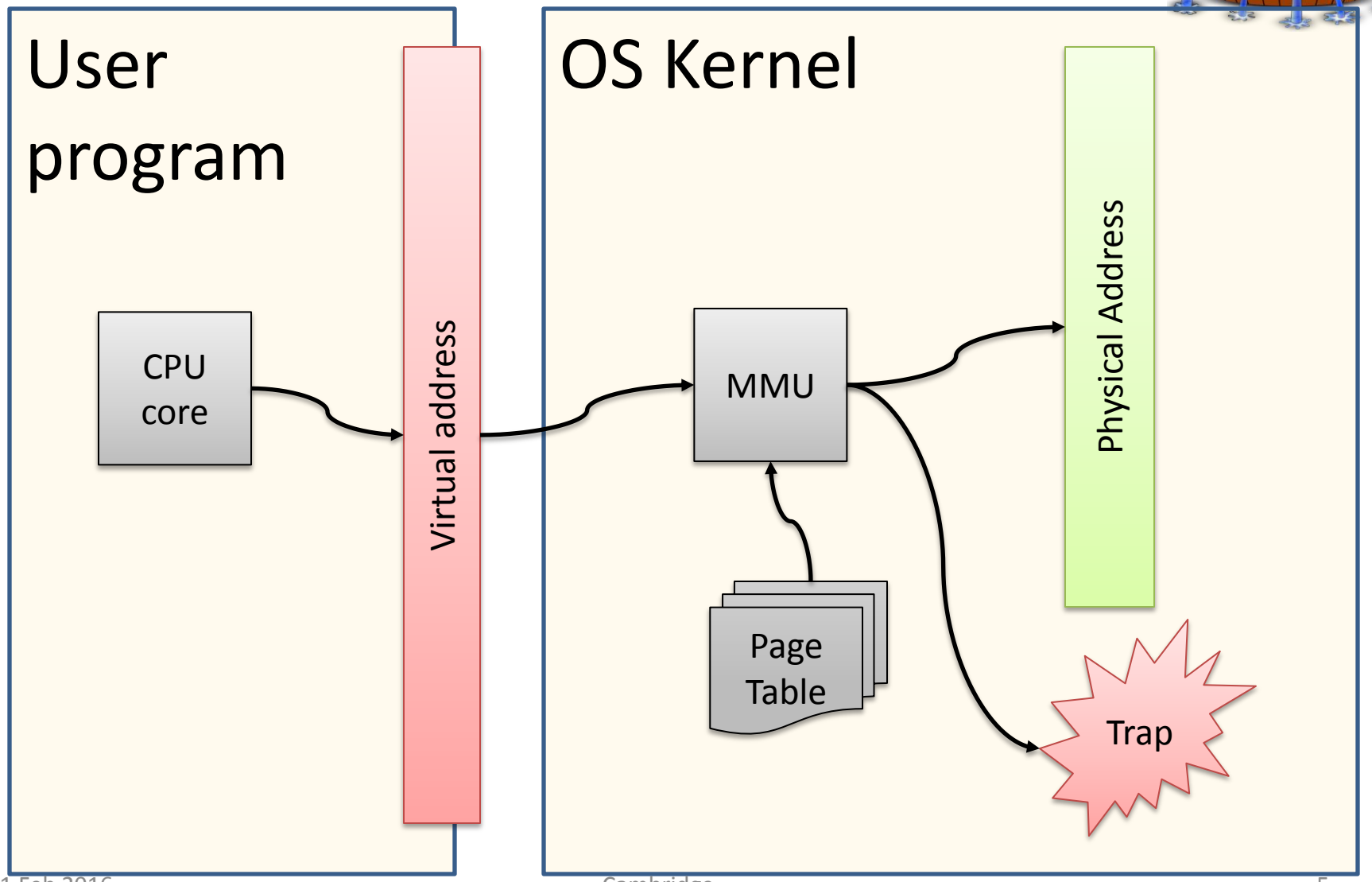
- Milestone: catching page faults
  - And faulting in new pages from the allocator
- In Linux: this is done in the kernel
- In Barrelfish (and other systems):
  - This is done by the application itself
  - Page fault $\Rightarrow$ application upcall
  - User library handles paging
- So, why?

MEMORY MANAGEMENT IN UNIX

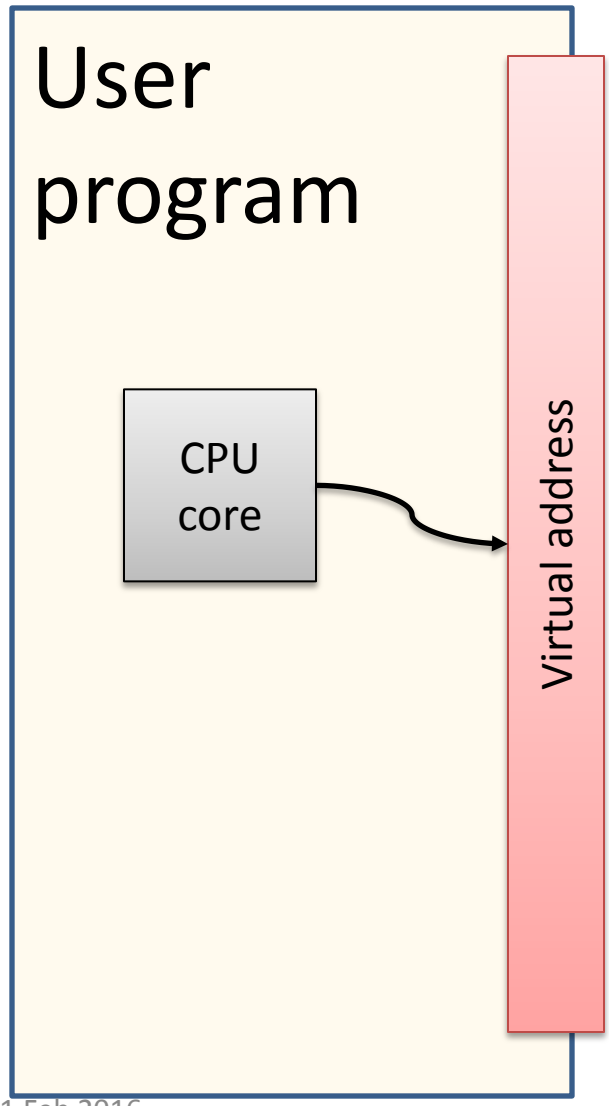
The classic Unix memory API



The classic Unix memory API



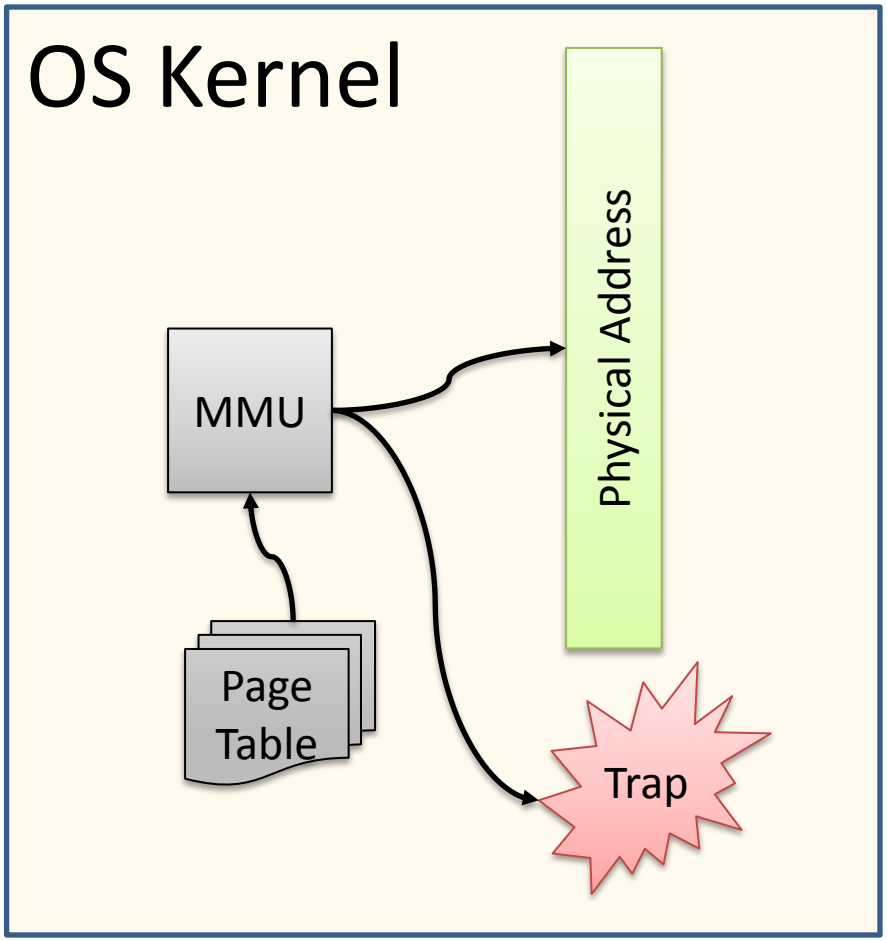
The classic Unix memory API



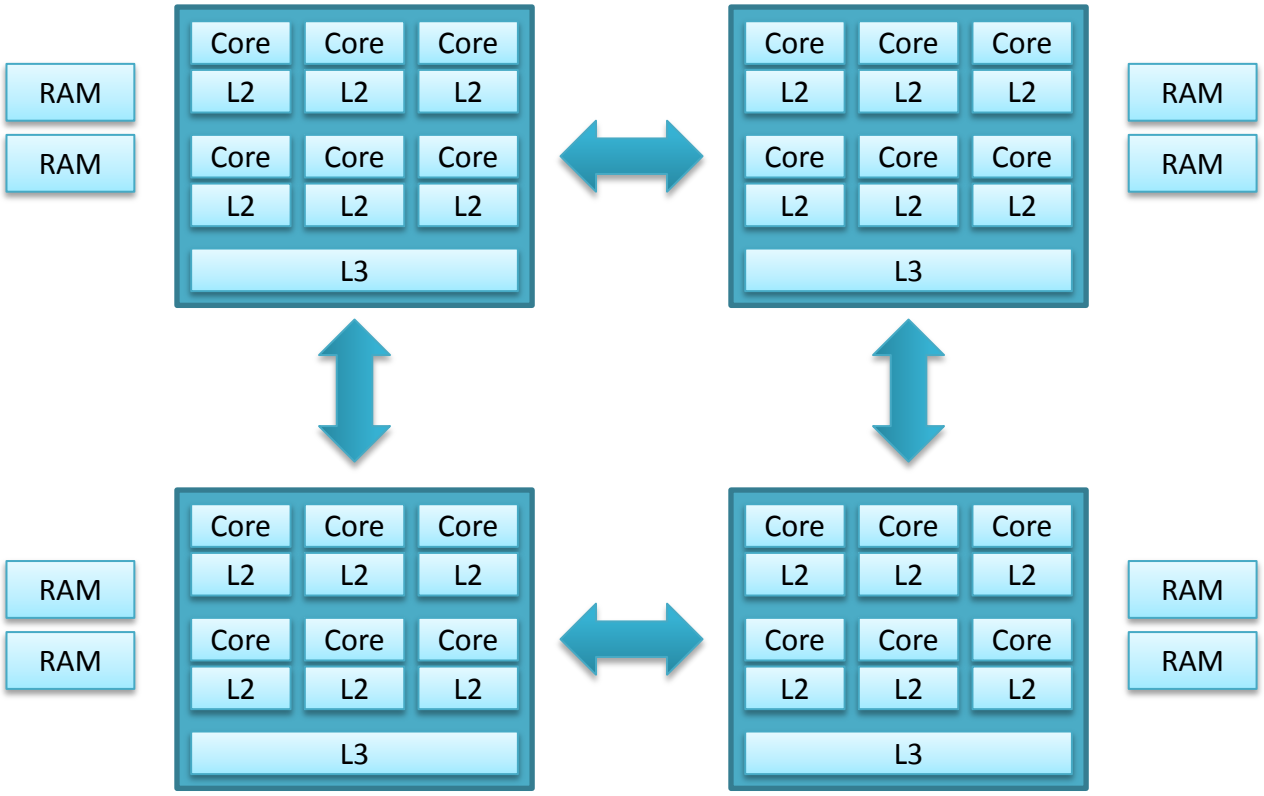
- Interface: Virtual Addresses
- Physical memory not visible
  - Demand paging
- Simple and Easy
- Inflexible and Opaque

Problem: Physical addresses matter

- NUMA regions
- Load balancing



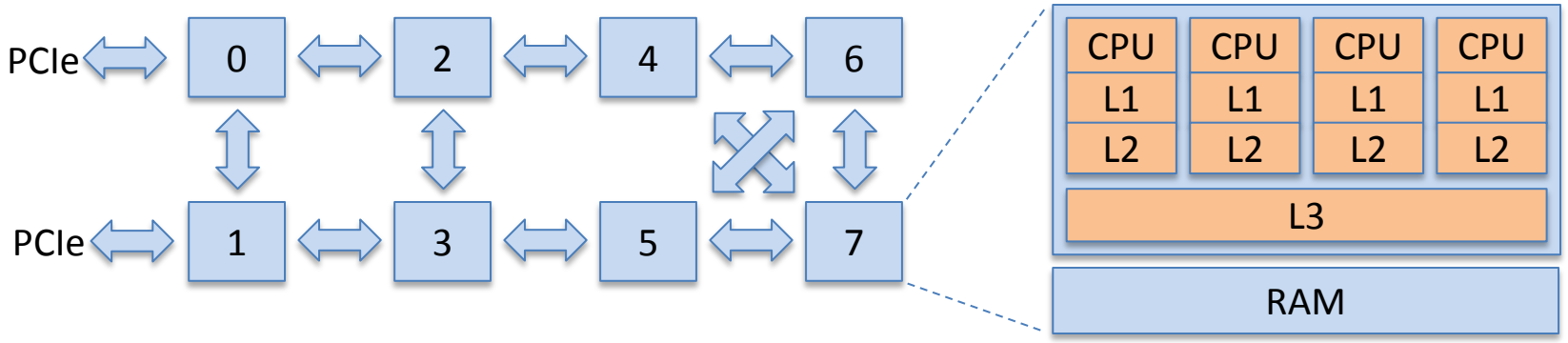
Cache-coherent multicore



AMD Istanbul: 6 cores, per-core L2, per-package L3

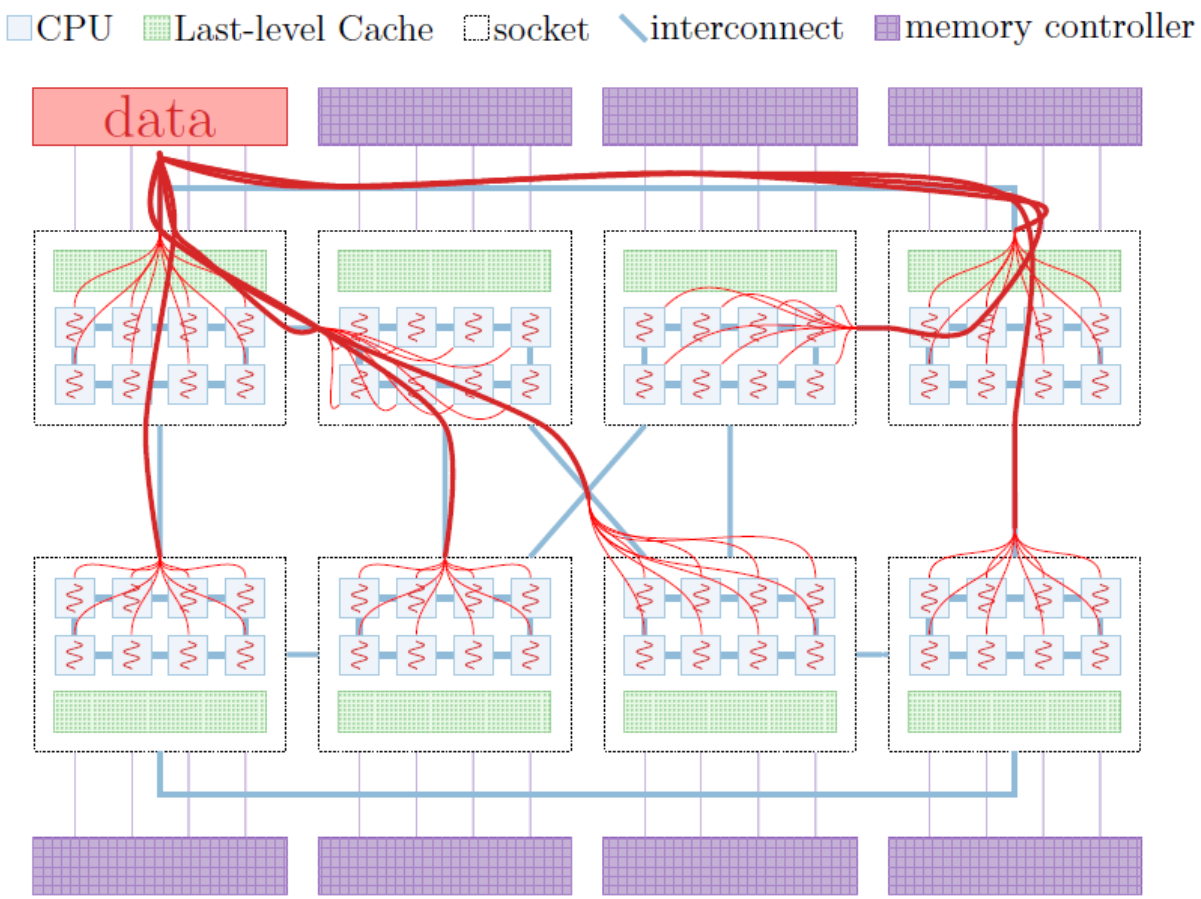
Latency

Example: 8 * quad-core AMD Opteron



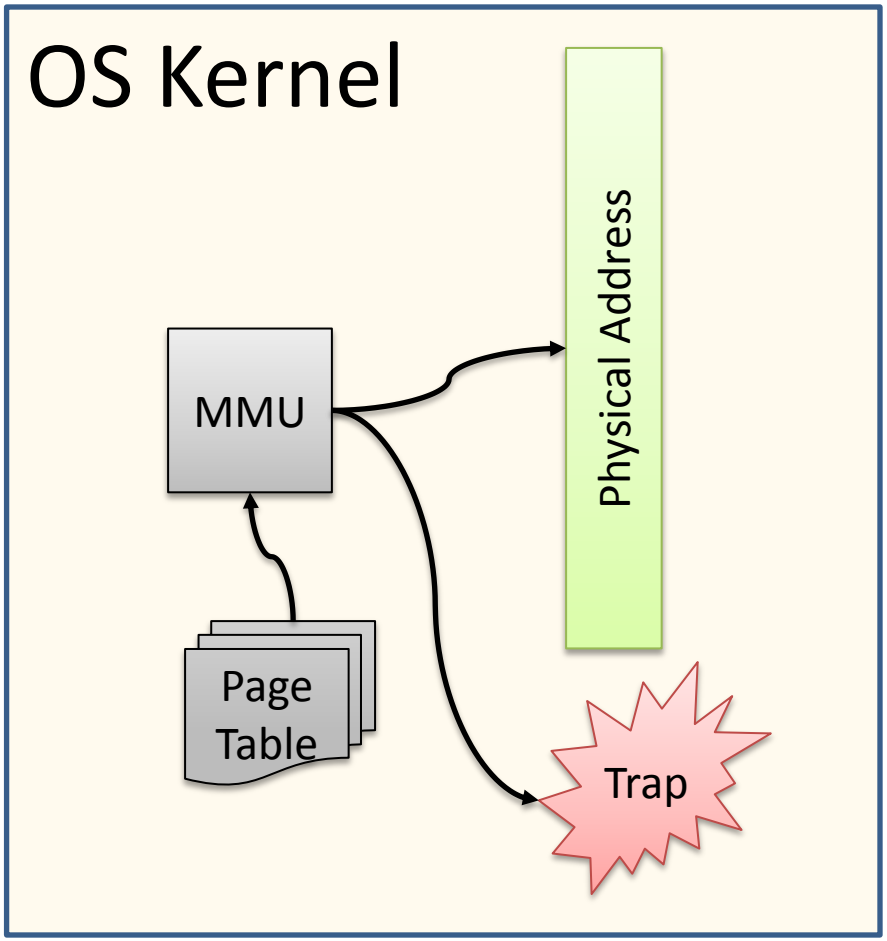
Access	cycles	normalized to L1	per-hop cost
L1 cache	2	1	-
L2 cache	15	7.5	-
L3 cache	75	37.5	-
Other L1/L2	130	65	-
1-hop cache	190	95	60
2-hop cache	260	130	70

Bandwidth contention



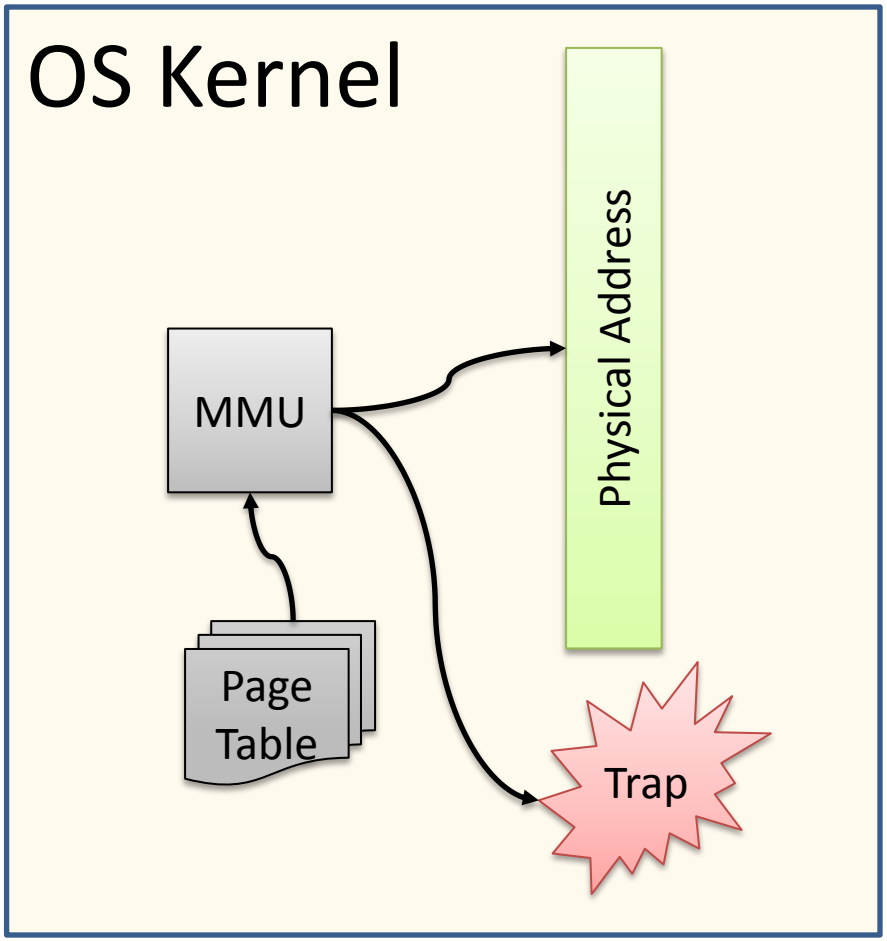
Problem: Physical addresses matter

- Pinned memory
- Devices or scratchpad
- Cache coloring
- Etc.

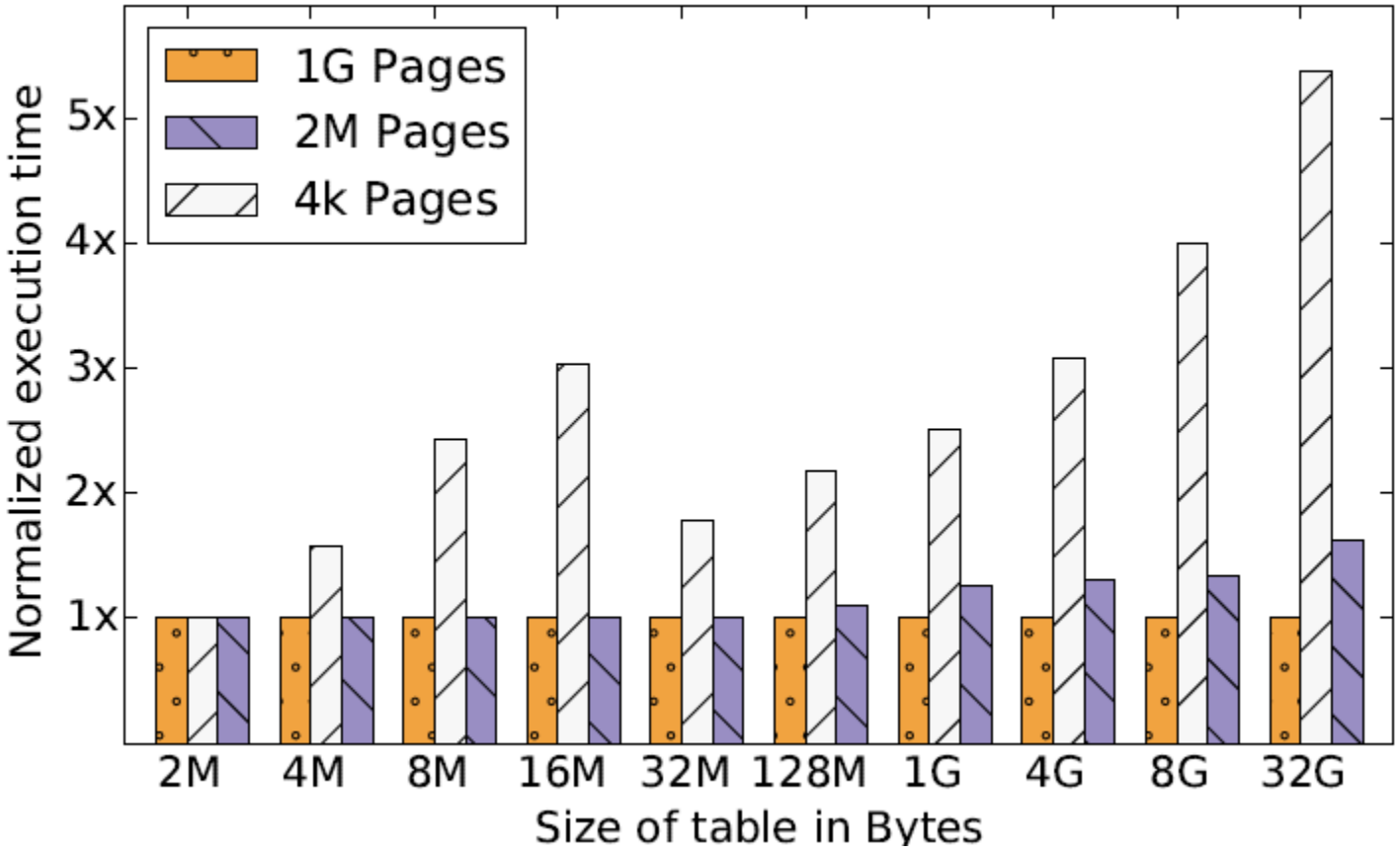


Problem: Translations matter

- Large and huge pages
- Intra-application protection

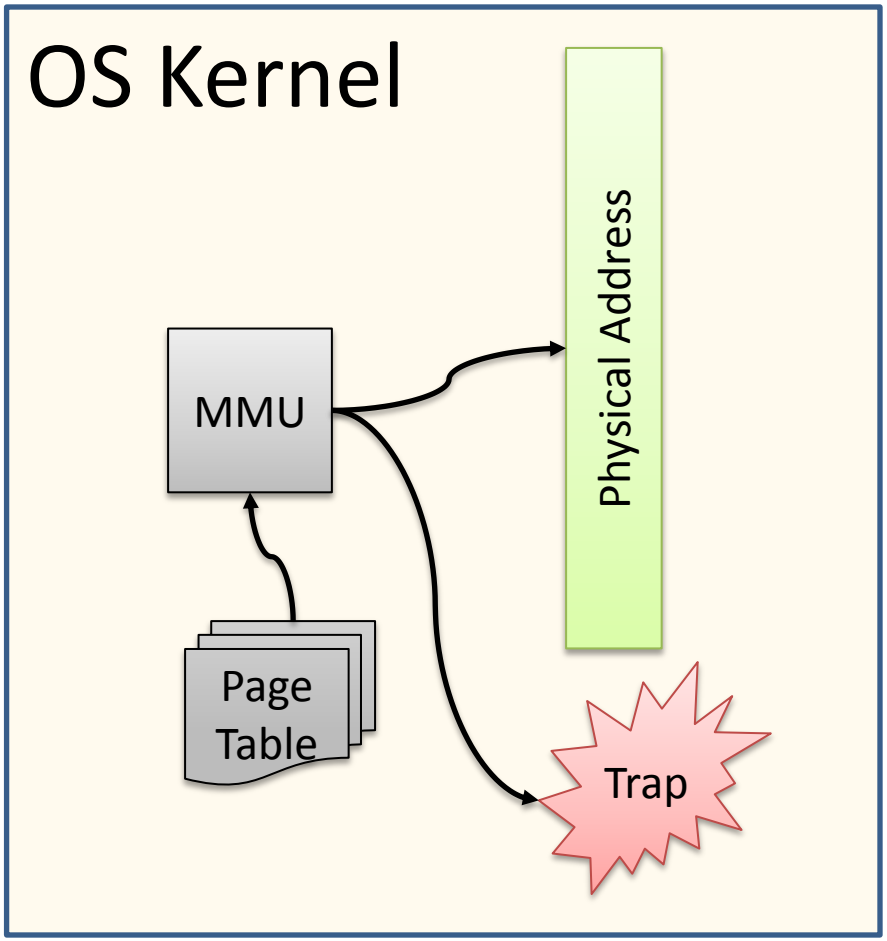


Large page benefits: GUPS



Problem: Traps matter

- Trapping for access interposition

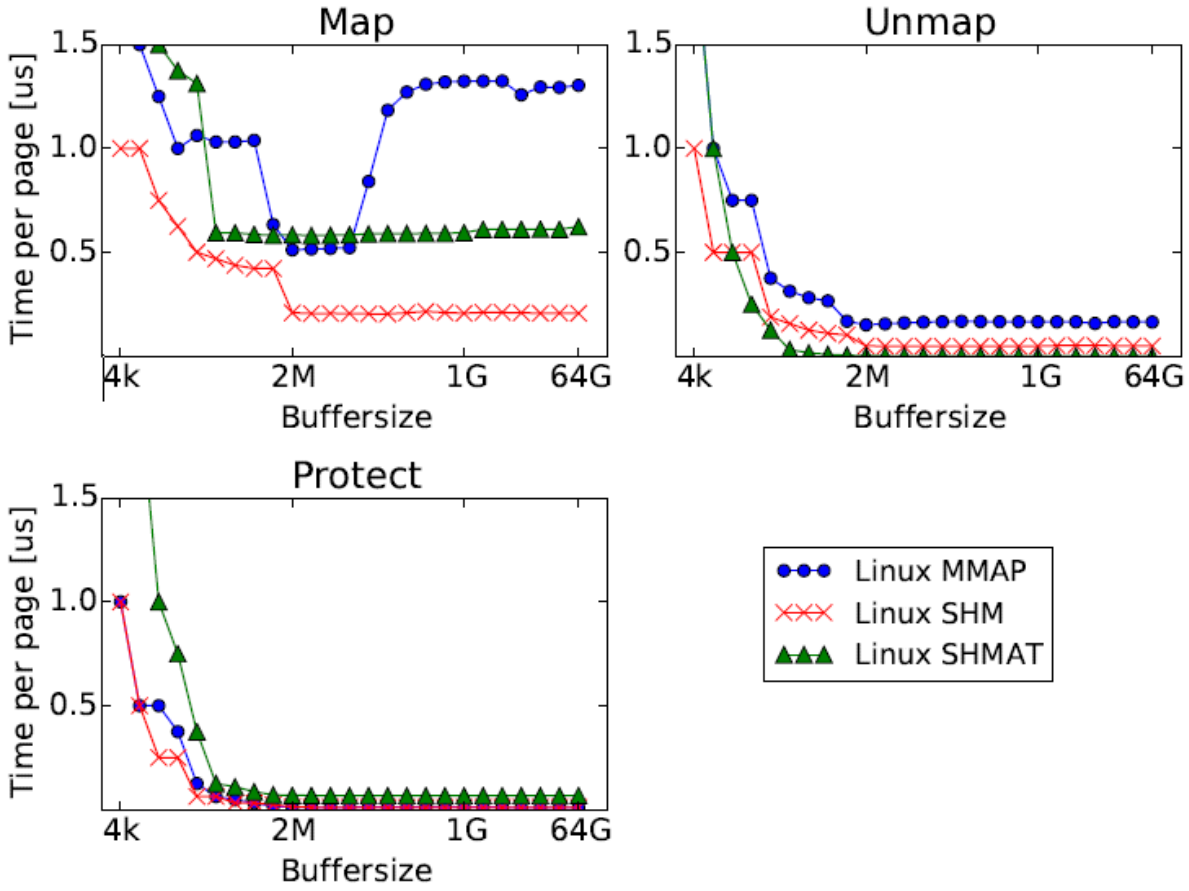


<Appel and Li paper about using traps for GC>

The Unix solution

- Evolving plethora of “holes” punched in the virtual address space
  - `mmap()` of regions
  - Large page support from pre-allocated pools
    - Wired on a per-process basis
  - Catching SEGV without translation information
  - `madvise()`
  - Etc., etc.
- All still refer to memory via virtual addresses

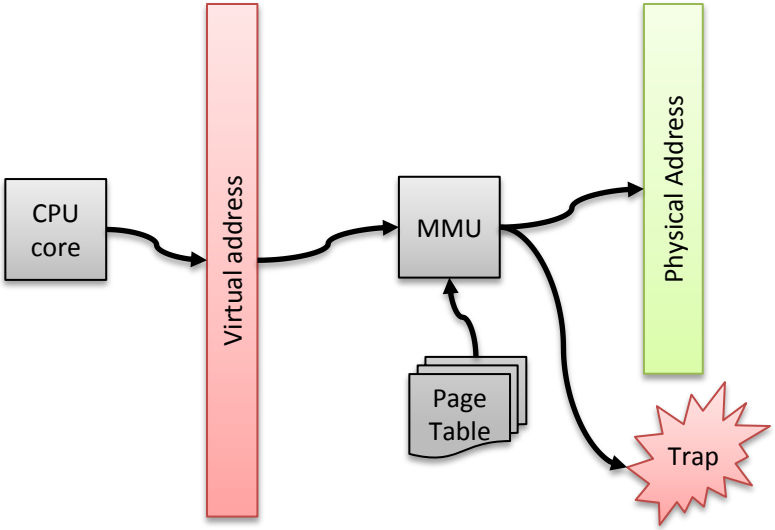
Too many interfaces – all virtual!



APPLICATION-LEVEL MEMORY MANAGEMENT

A step back

- Address translation hardware has many uses:
  - Demand paging
  - Relocation
  - Interposition
  - Replication
  - Etc.
- Interface only designed for Demand Paging
  - And who pages these days?



A New(ish) approach

- Applications instead:
  1. Manipulate physical memory regions
  2. Directly program translation hardware
  3. Receive translation faults
- Library provides virtual address space
  - Default emulates Unix
  - Complete flexibility modulo hardware

Key properties

Safety:

No process can install a page table which provides access to physical addresses for which is does not hold authorization.

Completeness:

A process can construct any safe *page table the hardware allows, subject to resource limits.*

User-managed virtual memory

Demand-paged virtual addresses	• If required
Distributed address space coordination	• Synchronization of page tables, etc.
Safe *direct* user access to MMU & TLB	• User-level page table construction • Reflection of page faults through upcalls
Physical Memory Management	• Capabilities • Address space algebra
Hardware	• MMU and TLB • First-level page fault handlers

1. Physical memory

- Applications refer to *physical* memory regions
  - Allocation, authorization, etc.

⇒ some kind of *descriptor*

  - We use Barrelfish capabilities
  - Might get by with POSIX file descriptors (?)

- Regions have *properties*
  - NUMA node, alignment, etc.

2. User-safe MMU access

- Applications directly handle physical memory
  $\Rightarrow$ library programs the MMU directly
  - Build page tables
  - Manage different page sizes
  - Handle page faults
  - Synchronize mappings across cores
- Security provided by capabilities
  - Any core can build a page table for any other

Capability operations

Capability	Type	Size
$Capref_a$	`VNode_x86_64_ptable`	4kB
$Capref_b$	`Frame`	4kB

$Capref_a \rightarrow$`install(`$Capref_b$, *slot* , *flags*`)`

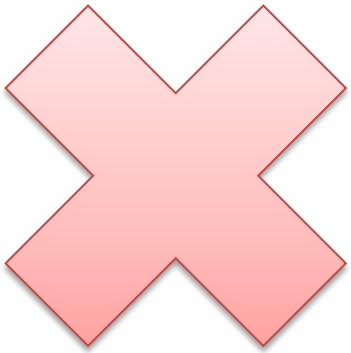
Map a 4kB page frame in the page table page

Which entry (0-511)

Capability operations

Capability	Type	Size
$Capref_a$	`VNode_x86_64_ptable`	4kB
$Capref_b$	**`VNode_x86_64_pdpt`**	4kB

$Capref_a \rightarrow \texttt{install}(Capref_b, slot, flags)$



Capability operations

Capability	Type	Size
$Capref_a$	`VNode_x86_64_ptable`	4kB
$Capref_b$	`Frame`	**16kB**

$Capref_a \rightarrow \texttt{install}(Capref_b, slot, flags)$

May work, depending on existing mappings and alignment

Capability operations

Capability	Type	Size
$Capref_a$	**`VNode_x86_64_pdir`**	4kB
$Capref_b$	`Frame`	**2MB**

$Capref_a \rightarrow \texttt{install}(Capref_b, slot, flags)$

Create a large page mapping

Capability operations

Capability	Type	Size
$Capref_a$	**`VNode_x86_64_pdpt`**	4kB
$Capref_b$	`Frame`	**1GB**

$Capref_a \rightarrow$`install(`$Capref_b$, *slot*, *flags*`)`

Create a huge page mapping

Capability operations

Capability	Type	Size
$Capref_a$	**VNode_x86_64_pml4**	4kB

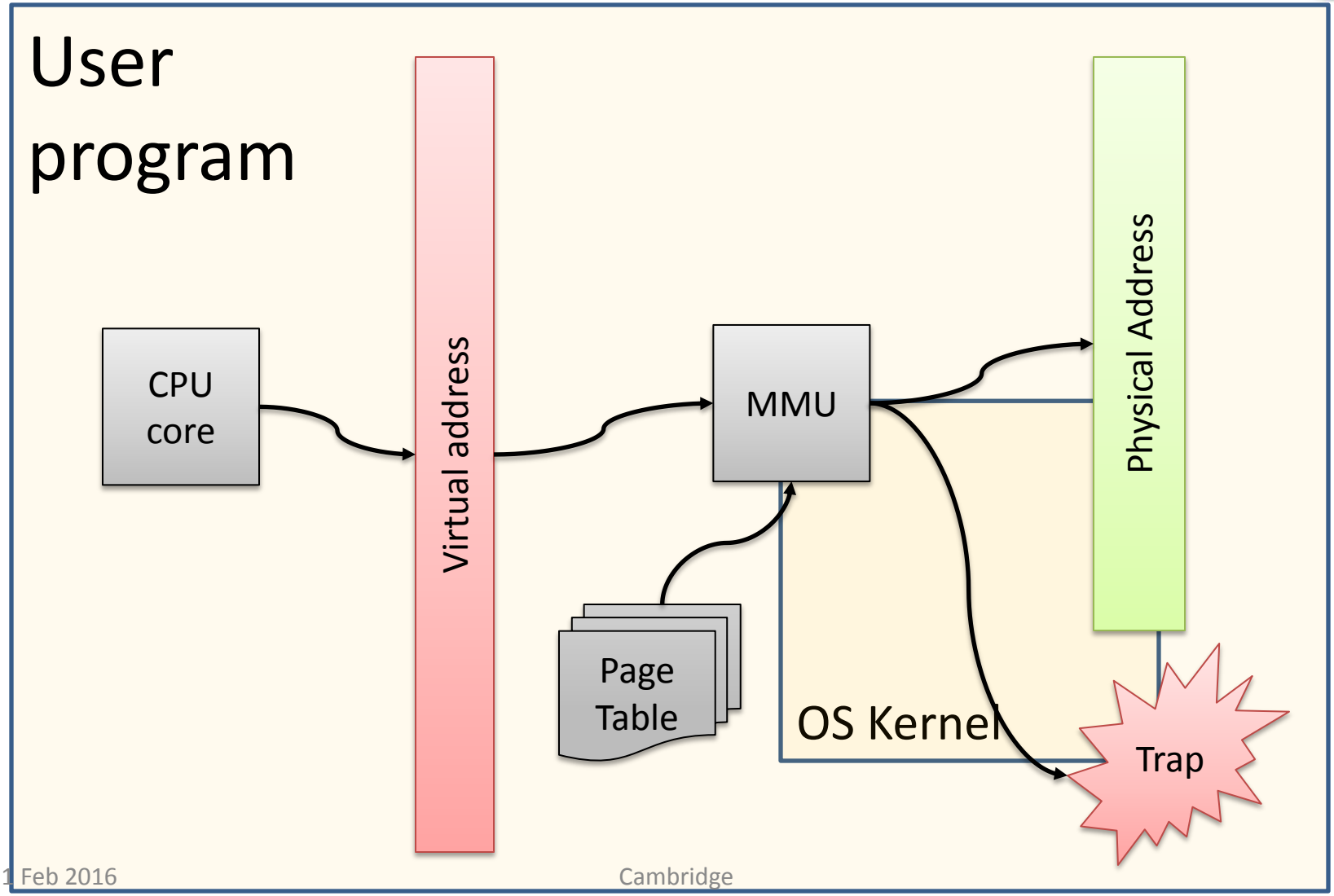
$Domain \rightarrow \texttt{switch}(Capref_a)$

Install new page table as our virtual address space

3. Translation faults

- Classic “self-paging” mechanism
  - Upcall to user space
  - Handled by application, not the kernel
- Integrated with Barrelfish process dispatch
  - Could use POSIX signals (?)
- Can implement demand paging
  - If you really want it – it’s a library

Barrelfish memory system

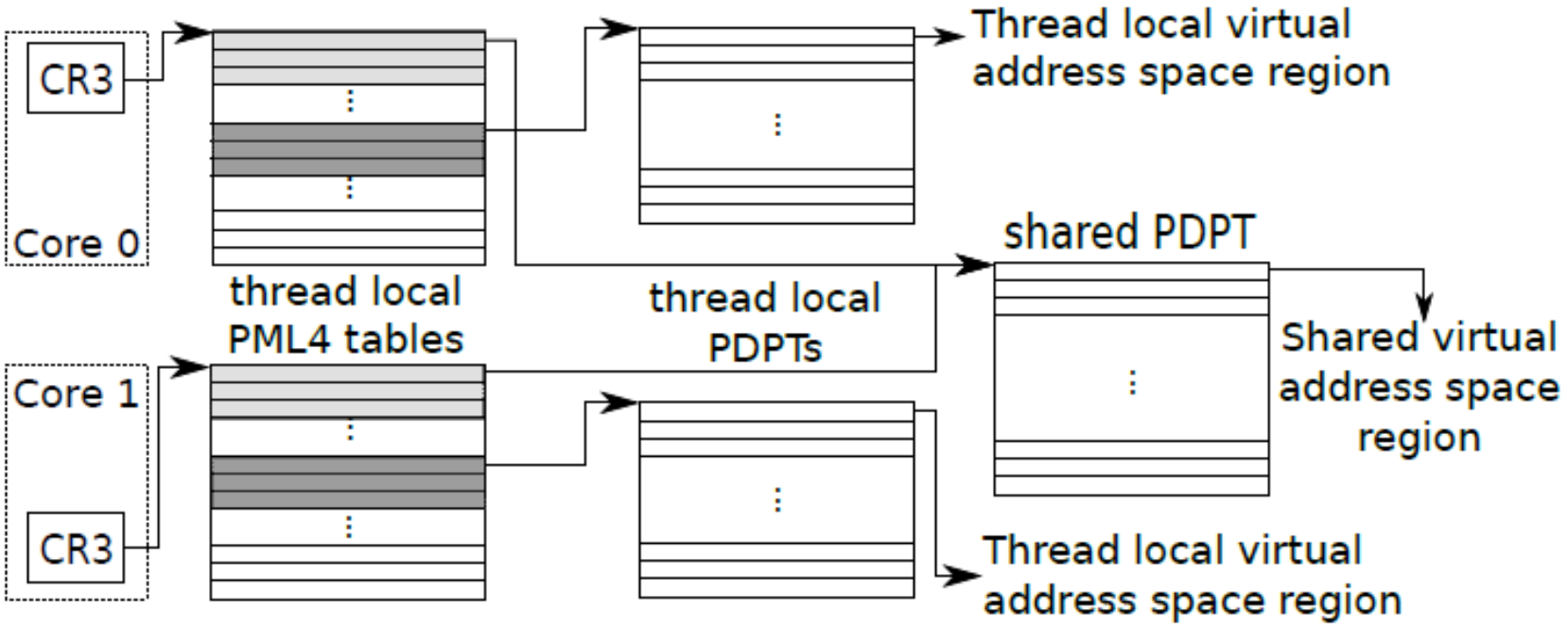


<If you really want to demand page, what do you need to do?>

- Different policies for different virtual regions?
- What can be paged, and what can not?
  - Clearly the paging library must be resident!
- Must block faulting thread, unblock it later
- Must track:
  - multiple faulting virtual addresses
  - corresponding physical addresses (via capabilities)

OTHER USES

Transparent replication



Transparent replication

- Page rank on Twitter graph using Green-Marl:

	Runtime (sec)	Std. error (sec)
No replication (default)	32.7	0.002
Software replication	28.1	0.001
Hardware replication	27.8	0.008

No changes to source code

EXPOSING ACCESS BITS

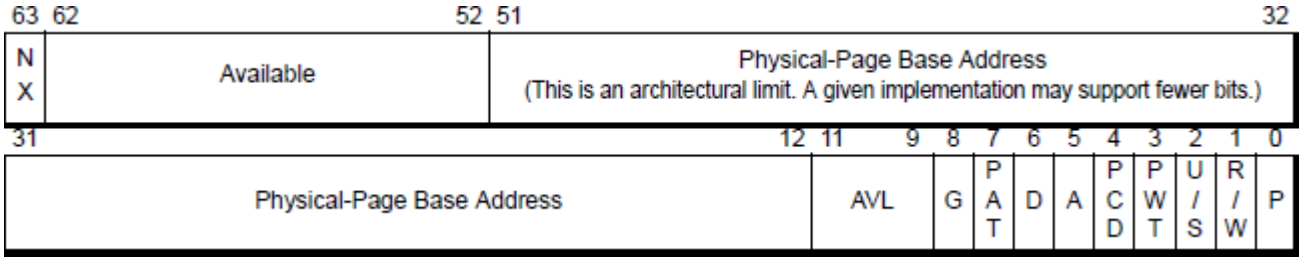
Capability operations

Capability	Type	Size
$Capref_a$	`VNode_x86_64_ptable`	4kB
$Capref_b$	**`VNode_x86_64_ptable`**	4kB

$Capref_a \rightarrow$`install(`$Capref_b$, *slot, flags*`)`

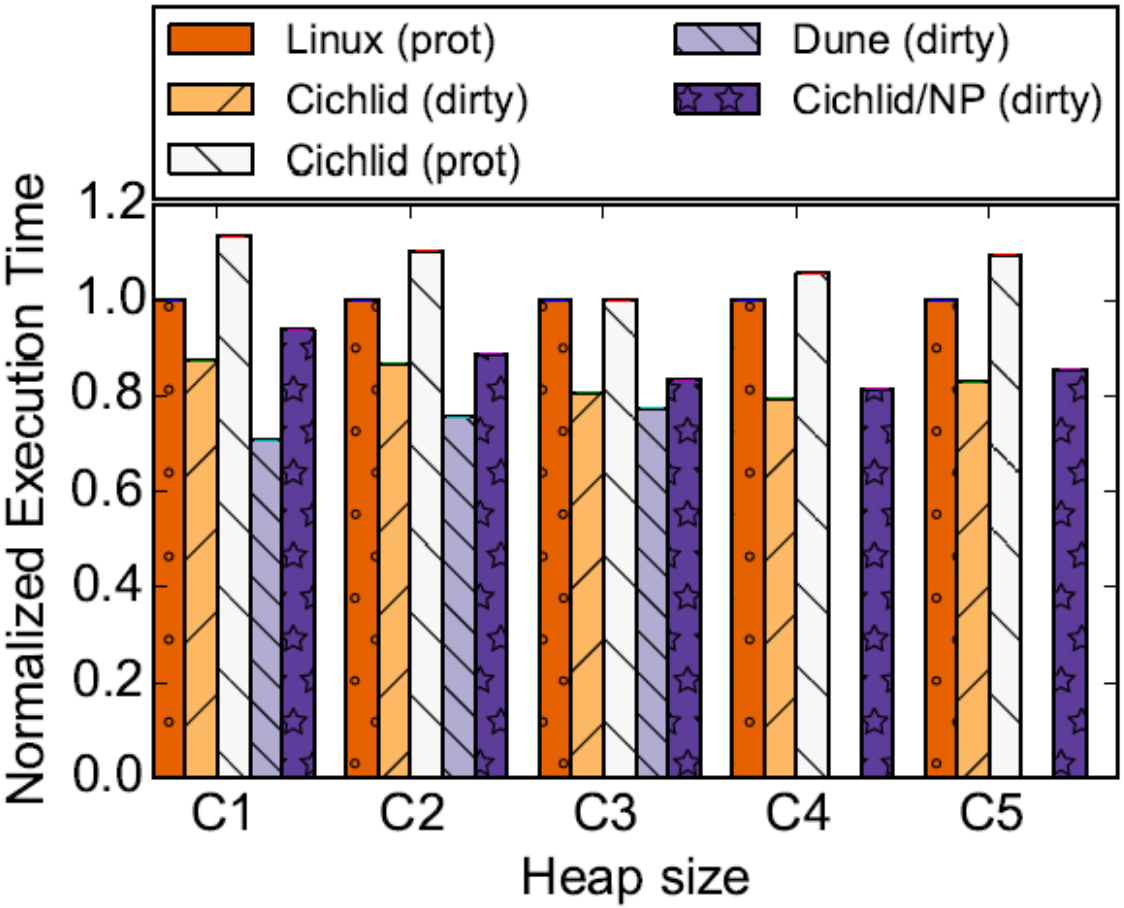
Map a page table read-only into virtual address space

Page Table Entry (level 1)

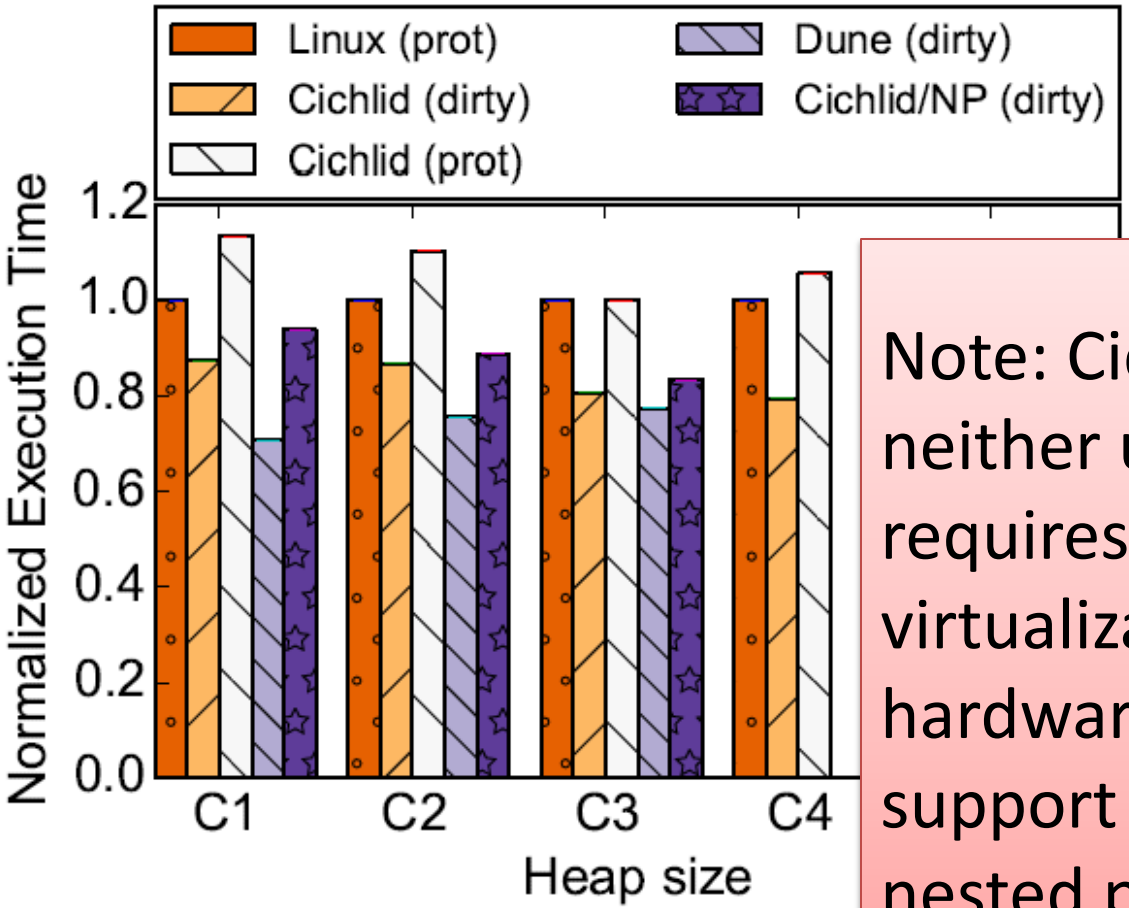


- **Physical Page base address:**
  40 most significant bits of physical page address (forces pages to be 4 KB aligned)
- **Avail:** available for system programmers
- **G:** global page (don't evict from TLB on task switch)
- **PAT:** Page-Attribute Table
- D: dirty (set by MMU on writes)
- A: accessed (set by MMU on reads and writes)
- **PCD:** cache disabled or enabled
- **PWT:** write-through or write-back cache policy for this page
- **U/S:** user/supervisor
- **R/W:** read/write
- **P:** page is present in physical memory (1) or not (0)

Garbage collection GCbench macrobenchmark



Garbage collection GCbench macrobenchmark

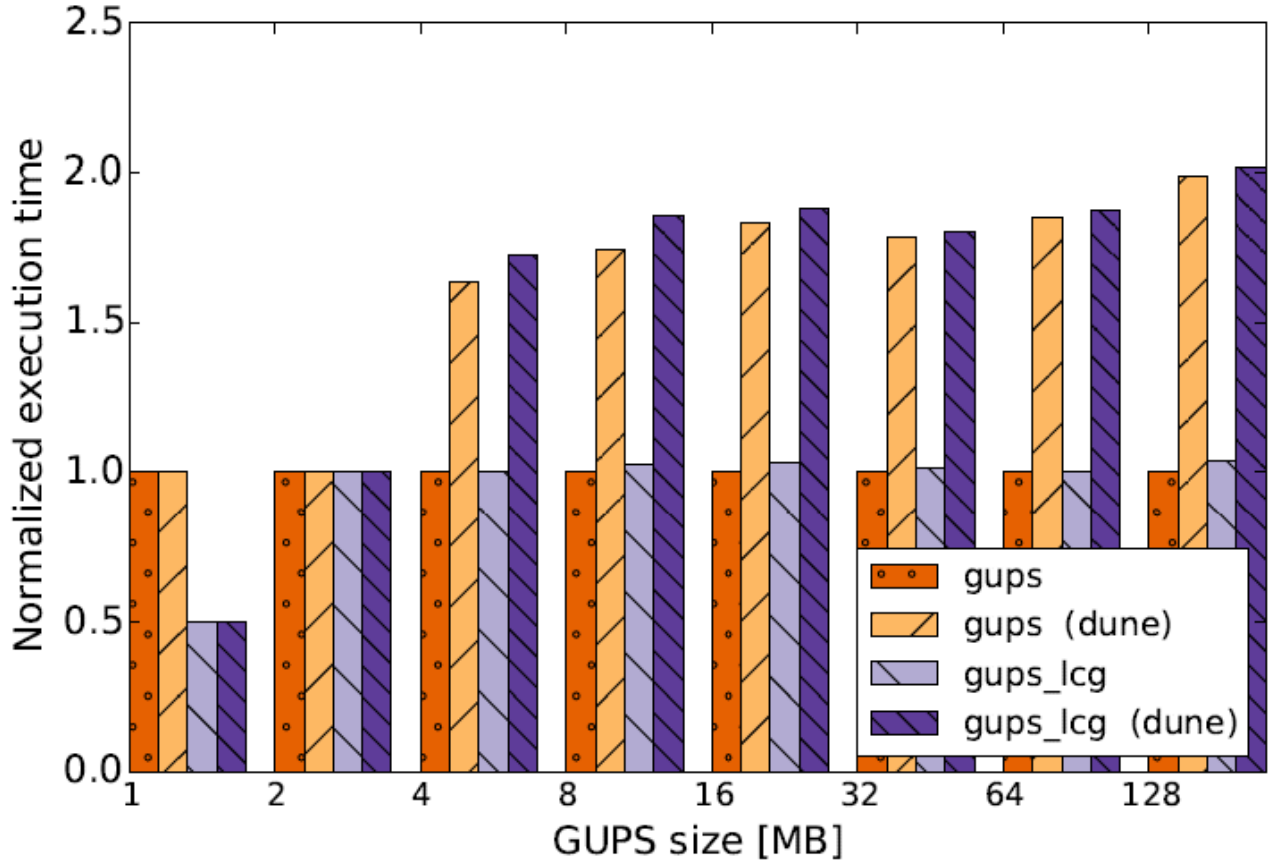


Note: Cichlid neither uses not requires virtualization hardware support for nested paging

EXPOSING ACCESS BITS

<something about Dune and virtualization hardware (Arrakis?)>

The cost of virtualization hardware



Does it work?

- Is at least as fast as Linux
  - For conventional VM operations
- Allows VM use-cases not possible in Linux
  - Dynamic large pages, partial replication, access/dirty bits, cache coloring, etc.
- Doesn't need virtualization hardware
  - But could be extended to use it

FURTHER READING

Further reading

- Kieran Harty and David R. Cheriton. 1992. Application-controlled physical memory using external page-cache management. In Proceedings of the fifth international conference on Architectural support for programming languages and operating systems (ASPLOS V), Richard L. Wexelblat (Ed.). ACM, New York, NY, USA, 187-197. http://dx.doi.org/10.1145/143365.143511
- Andrew W. Appel and Kai Li. 1991. Virtual memory primitives for user programs. In Proceedings of the fourth international conference on Architectural support for programming languages and operating systems (ASPLOS IV). ACM, New York, NY, USA, 96-107. http://dx.doi.org/10.1145/106972.106984
- M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. 1997. Application performance and flexibility on exokernel systems. In Proceedings of the sixteenth ACM symposium on Operating systems principles (SOSP '97), William M. Waite (Ed.). ACM, New York, NY, USA, 52-65. http://dx.doi.org/10.1145/268998.266644

Further reading

- Steven M. Hand. 1999. Self-paging in the Nemesis operating system. In Proceedings of the third symposium on Operating systems design and implementation (OSDI '99). USENIX Association, Berkeley, CA, USA, 73-86. https://www.usenix.org/publications/library/proceedings/osdi99/full_papers/hand/hand.pdf
- D. R. Engler, S. K. Gupta, and M. F. Kaashoek. 1995. AVM: application-level virtual memory. In Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V) (HOTOS 95). IEEE Computer Society, Washington, DC, USA, 72-. https://pdos.csail.mit.edu/~engler/avm.ps
- Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. 2008. Kernel design for isolation and assurance of physical memory. In Proceedings of the 1st workshop on Isolation and integration in embedded systems (IIES '08), Michael Engel and Olaf Spinczyk (Eds.). ACM, New York, NY, USA, 35-40. http://dx.doi.org/10.1145/1435458.1435465
- Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: safe user-level access to privileged CPU features. In Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI'12). USENIX Association, Berkeley, CA, USA, 335-348. https://www.usenix.org/system/files/conference/osdi12/osdi12-final-117.pdf