NVDIMM Overview

Technology, Linux, and Xen

Who am I?

What are NVDIMMs?

- A standard for allowing NVRAM to be exposed as normal memory
- Potential to dramatically change the way software is written

But..

- They have a number of surprising problems to solve
- Incomplete specifications and confusing terminology

Goals

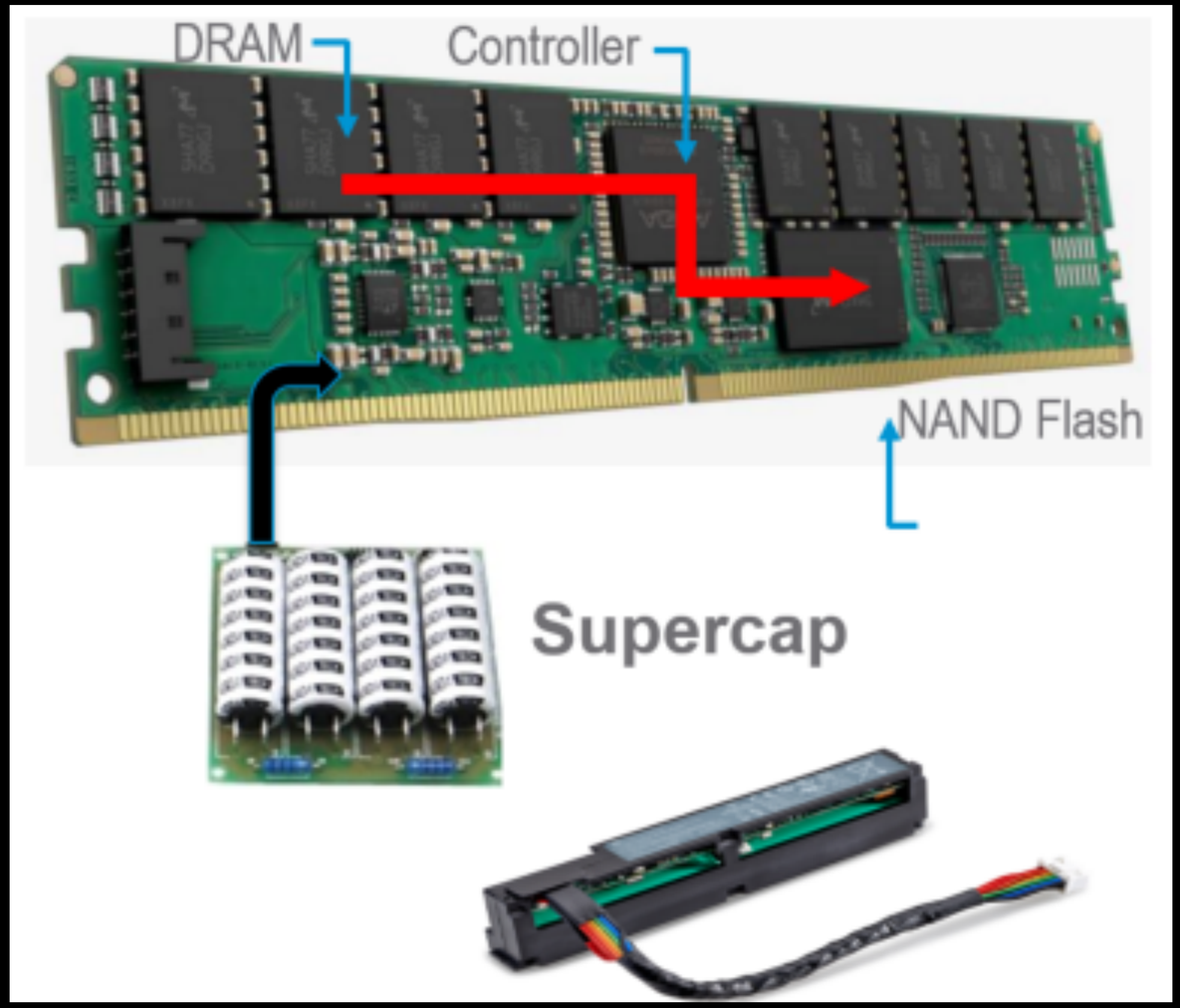
- Give an overview of NVDIMM concepts, terminology, and architecture
- Introduce some of the issues they bring up WRT operating systems
- Introduce some of the issues faced wrt XEN

Caveats

- I’m not an expert
  - (So some of this information may be not be 100%)
- The situation is still developing
  - (so some of this information may be outdated)

Current Technology

- Currently available: NVDIMM-N
  - DRAM with flash + capacitor backup
  - Strictly more expensive than a normal DIMM with the same storage capacity

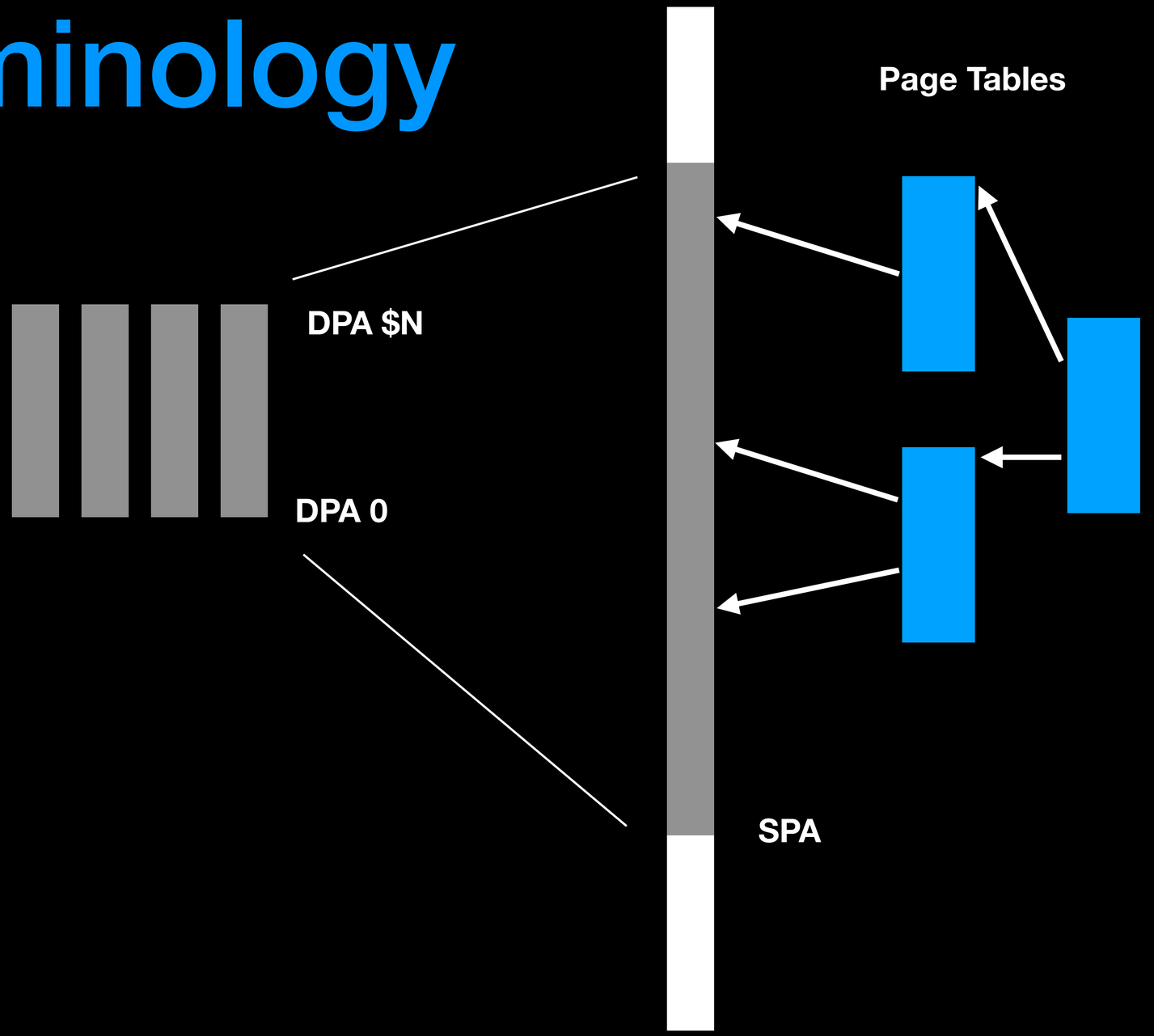


Coming soon: Terabytes

- Almost as cheap as disk
- Almost as fast as DRAM

Terminology

- Physical DIMM
- DPA: DIMM Physical Address
- SPA: System Physical Address

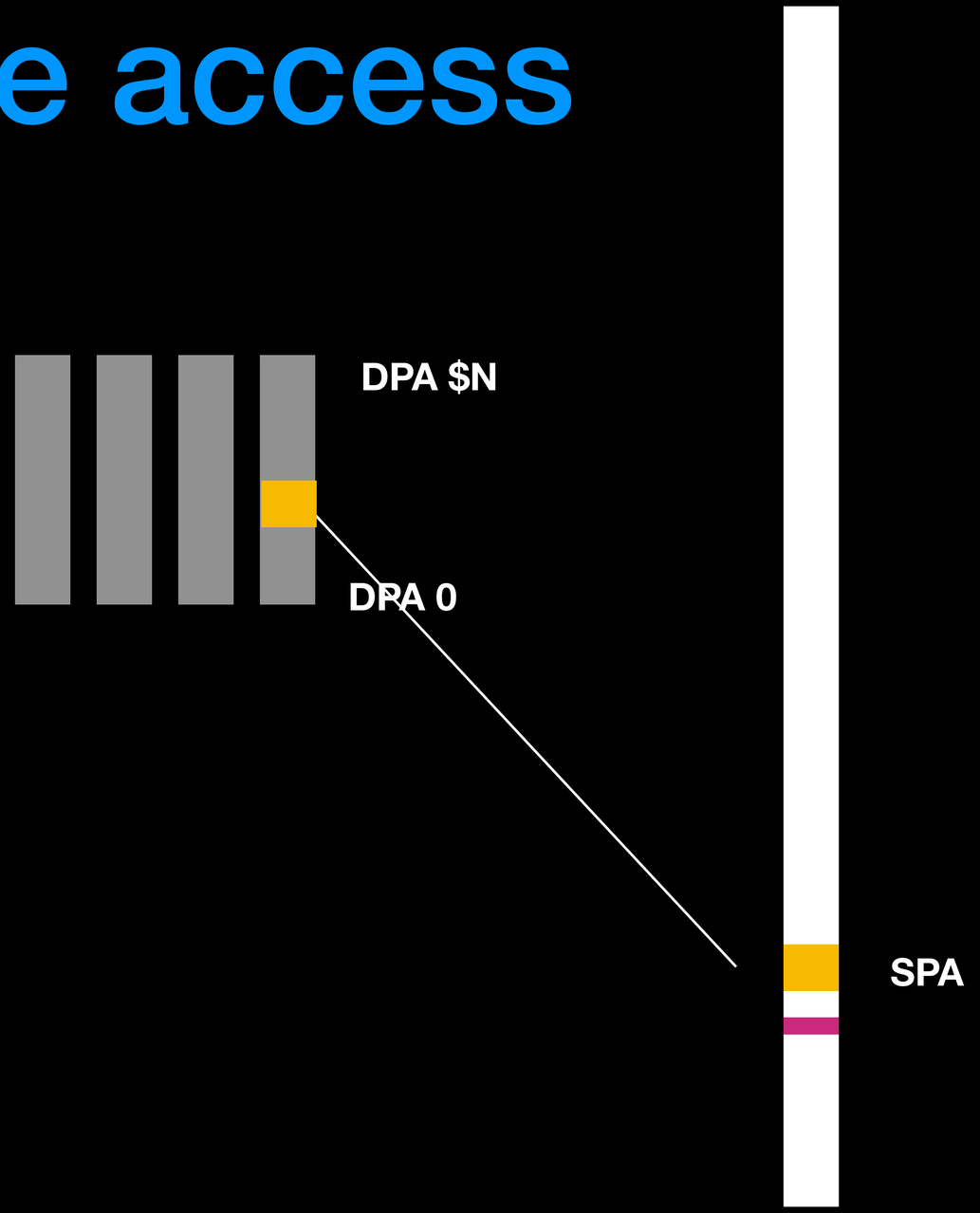


PMEM: RAM-like access

- DRAM-like access
- 1-1 Mapping between DPA and SPA
- Interleaved across DIMMs
- Similar to RAID-0
  - Better performance
  - Worse reliability

PBLK: Disk-like access

- Disk-like access
  - Control region
  - 8k Data “window”
  - One window per NVDIMM device
- Never interleaved
- Useful for software RAID
- Also useful when SPA space < NVRAM size
  - 48 address bits (256TiB)
  - 39 physical bits (0.5 TiB)



How is this mapping set up?

- Firmware sets up the mapping at boot
  - May be modifiable using BIOS / vendor-specific tool
- Exposes information via ACPI

ACPI: Devices

- NVDIMM Root Device
  - Expect one per system
- NVDIMM Device
  - One per NVDIMM
  - Information about size, manufacture, &c

ACPI: NFIT Table

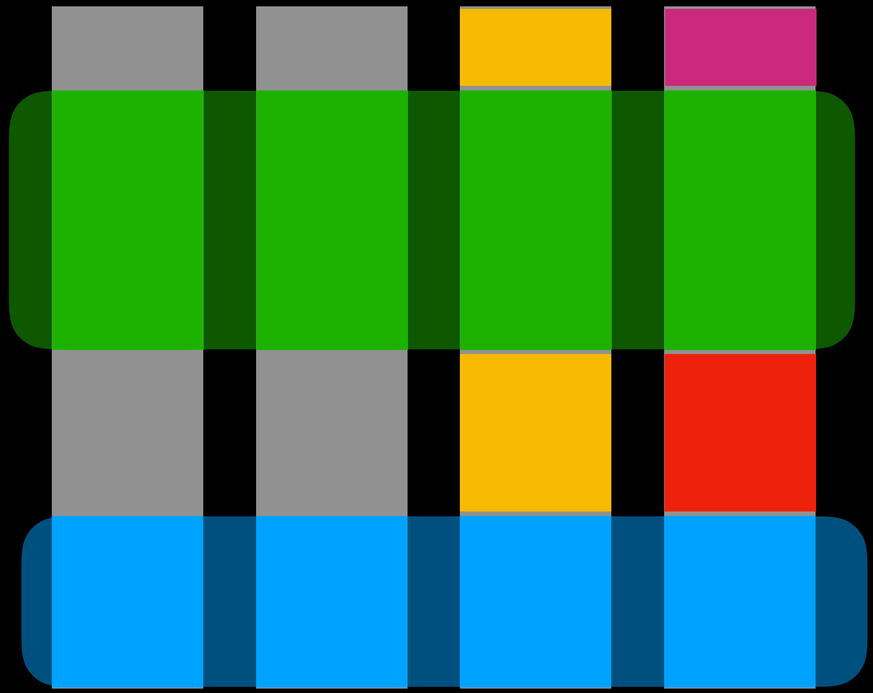
- NVDIMM Firmware Interface Table
- PMEM information:
  - SPA ranges
  - Interleave sets
- PBLK information
  - Control regions
  - Data window regions

Practical issues for using NVDIMM

- How to partition up the NVRAM and share it between operating systems
- Knowing the correct way to access each area (PMEM / PBLK)
- Detecting when interleaving / layout has changed

Dividing things up: Namespaces

- “Namespace”: Think partition
- PMEM namespace and interleave sets
- PBLK namespaces
- “Type UUID” to define how it’s used
  - think DOS partitions: “Linux ext2/3/4”, “Linux Swap”, “NTFS”, &c

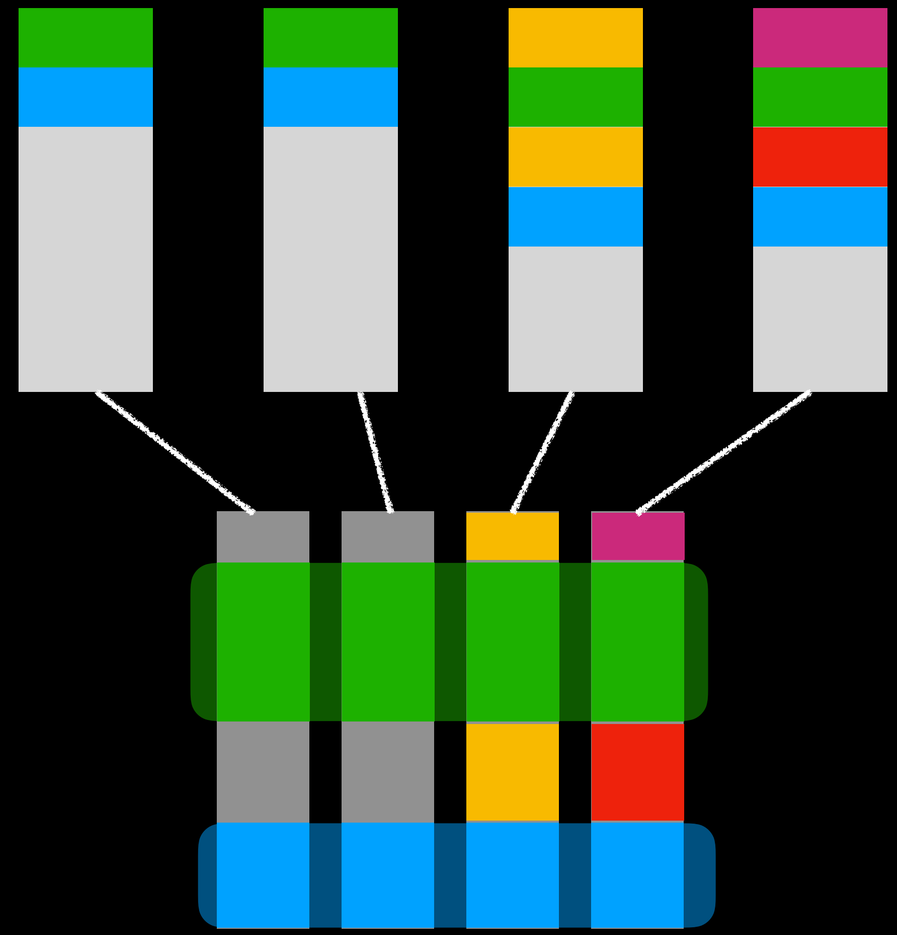


How do we store namespace information?

- Reading via PMEM and PBLK will give you different results
- PMEM Interleave sets may change across reboots
- So we can’t store it inside the visible NVRAM

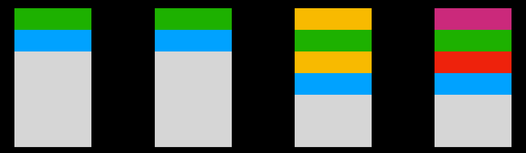
Label area: Per-NVDIMM storage

- One “label area” per NVDIMM device (aka physical DIMM)
- Label describes a single contiguous DPA region
- Namespaces made out of labels
- Accessed via ACPI AML methods
  - Pure read / write

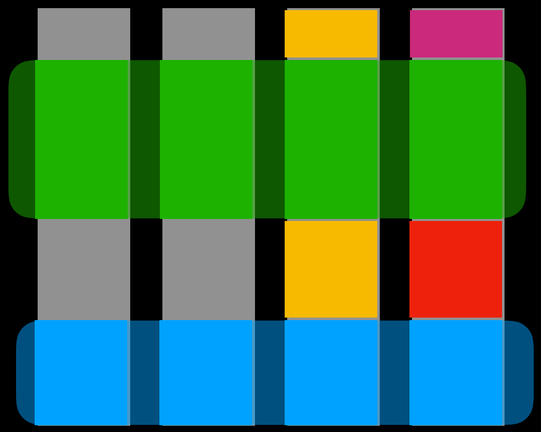


How an OS Determines Namespaces

- Read ACPI NFIT to determine
  - How many NVDIMMs you have
  - Where PMEM is mapped
- Read label area for each NVDIMM
- Piece together the namespace described
- Double-check interleave sets with the interleave sets (from NFIT table)
- Access PMEM regions by offsets in SPA sets (from NFIT table)
- Access PBLK regions by programming control / data windows (from NFIT table)



Label Areas



SPA

Key points

- “Namespace”: Partition
- “Label area”: Partition table
- NVDIMM devices / SPA ranges / etc defined in ACPI static NFIT table
- Label area accessed via ACPI AML methods

NVDIMMs in Linux

ndctl

- Create / destroy namespaces
- Four modes
  - raw
  - sector
  - fsdax
  - devdax

The ideal interface

- Guest processes map a normal file, and magically get permanent storage

The obvious solution

- Make a namespace into a block device
- Put a filesystem on it
- Have mmap() map a file to the NVRAM memory directly

Issue: Sector write atomicity

- Disk sector writes are atomic: all-or-nothing
- memcpy() can be interrupted in the middle
- Block Translation Table (BTT): an abstraction that guarantees write atomicity
- ‘sector mode’
- But this means mmap() needs a separate buffer

Issue: Page struct

- To keep track of userspace mappings, Linux needs a ‘page struct’
- 64 bytes per 4k page
  - 1 TiB of PMEM requires 7.85GiB of page array
- Solution: Use PMEM to store a ‘page struct’
- Use a superblock to designate areas of the namespace to be used for this purpose (allocated on namespace creation)

Issue: Filesystems and block location

- Filesystems want to be able to move blocks around
- Difficult interactions between write() system call (DMA)

Issue: Interactions with the page cache

Mode summary: Raw

- Block mode access to full SPA range
- No support for namespaces
- Therefore, no UUIDs / superblocks; page structs must be stored in main memory
- Supports DAX

Mode summary: Sector

- Block mode with BTT for sector atomicity
- Supports namespaces
- No DAX / direct mmap() support

Mode summary: fsdax

- Block mode access to a namespace
- Supports page structs in main memory, or in the namespace
  - Must be chosen at time of namespace creation
- Supports filesystems with DAX
  - But there’s some question about the safety of this

Mode summary: devdax

- Character device access to namespace
- Does not support filesystems
- page structs must be contained within the namespace
- Supports mmap()
- “No interaction with kernel page cache”
- Character devices don’t have a “size”, so you have to remember

Summary

- Four ways of mapping with different advantages and disadvantages
- Seems clear that Linux is still figuring out how best use PMEM

NVDIMM, Xen, and dom0

Issue: Xen and AML

- Reading the label areas can only be done via AML
- Xen cannot do AML
  - ACPI spec requires only a single entity to do AML
  - That must be domain 0

Issue: struct page

- In order to track mapping to guests, the hypervisor needs a struct page
  - “frametable”
  - 32 or 40 bits

Issue: RAM vs MMIO

- Dom0 is free to map any SPA
  - RAM: Page reference counts taken
  - MMIO: No reference counts taken
- Want to ref-count dom0 mappings to be sure we can safely pass PMEM to untrusted guests

Circular dependency

- To do refcounting, we need page structs
- To do page structs we need some ‘scratch’ area of PMEM
- To know where namespaces are we need AML
- To interpret a superblock dom0 needs to map it
- To map it, we want refcounts…

Simple solution

- Allocate a full namespace just for Xen
- Have dom0 read the label areas and pass the namespace range to Xen before accessing any PMEM ranges (enforced by Xen)

Questions?