ISE SRA

Leibniz Universität Hannover

Refreshing Memories

Non-Volatility in Volatile Address Spaces

Stefan Naumann, Daniel Lohmann

24 September 2020

NVRAM and its Use Cases

- Byte-adressable, low latency, less energy consuming, persistent
- Commercially available
- Large and expected to be very cheap

Persistence with File Systems

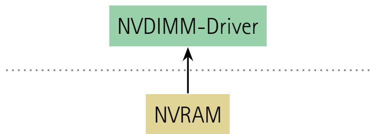
Huge DRAM

Huge Persistent Best-Effort Cache

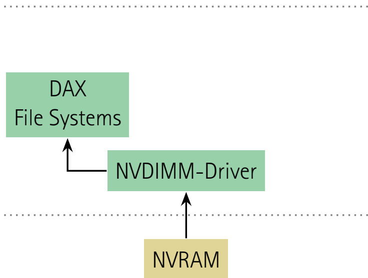
Programming with NVRAM: PMDK

NVRAM

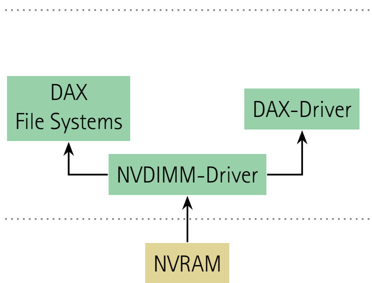
Programming with NVRAM: PMDK



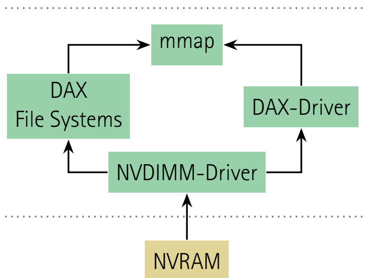
Programming with NVRAM: PMDK



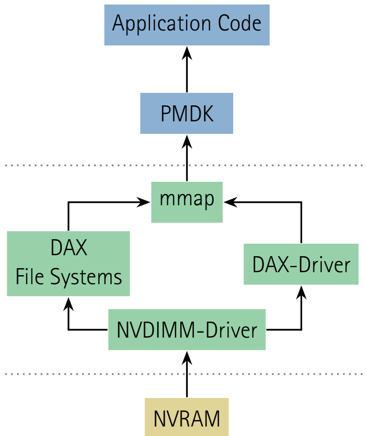
Programming with NVRAM: PMDK



Programming with NVRAM: PMDK



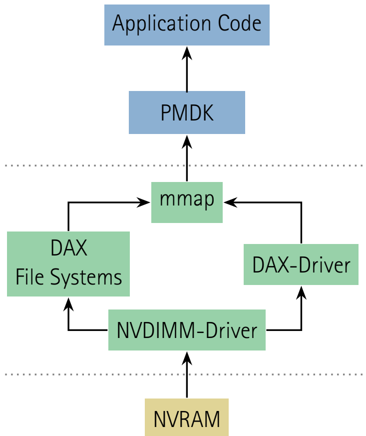
Programming with NVRAM: PMDK



Persistent Memory Development Kit (PMDK):

- Object pool management, including memory allocators

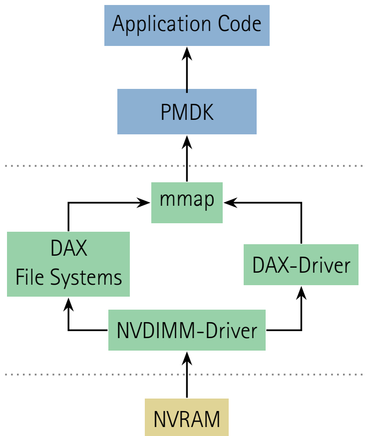
Programming with NVRAM: PMDK



Persistent Memory Development Kit (PMDK):

- Object pool management, including memory allocators
- Crash consistency by transactions

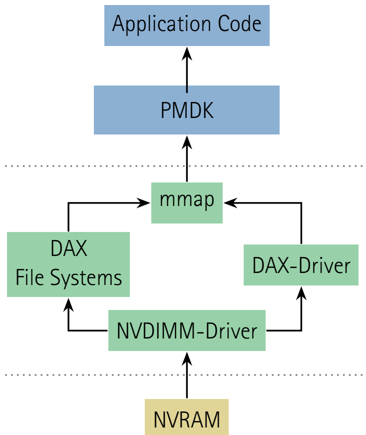
Programming with NVRAM: PMDK



Persistent Memory Development Kit (PMDK):

- Object pool management, including memory allocators
- Crash consistency by transactions
- Bindings to several languages

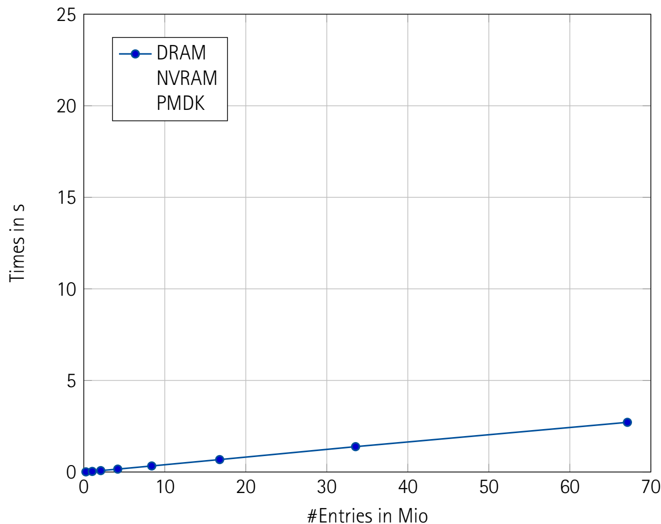
Programming with NVRAM: PMDK



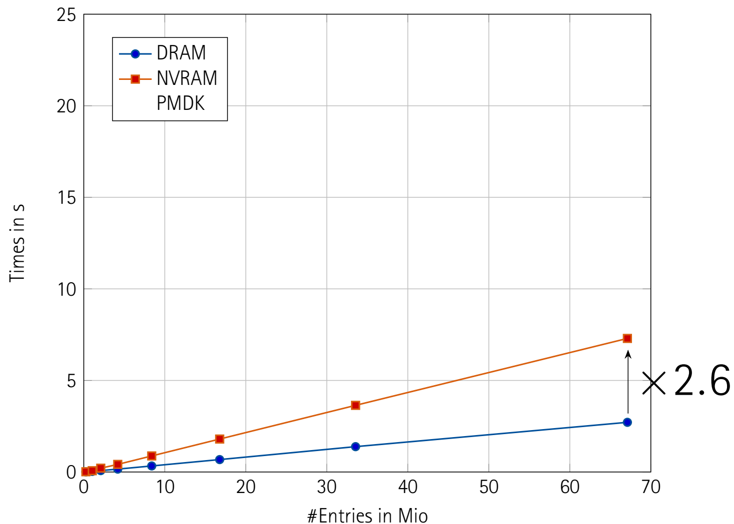
Persistent Memory Development Kit (PMDK):

- Object pool management, including memory allocators
- Crash consistency by transactions
- Bindings to several languages
- Cons:
  - Uses an **own pointer format**
  - Reimplements libc-calls for its own pointers
  - SLOC: 126k & 115k C++ bindings

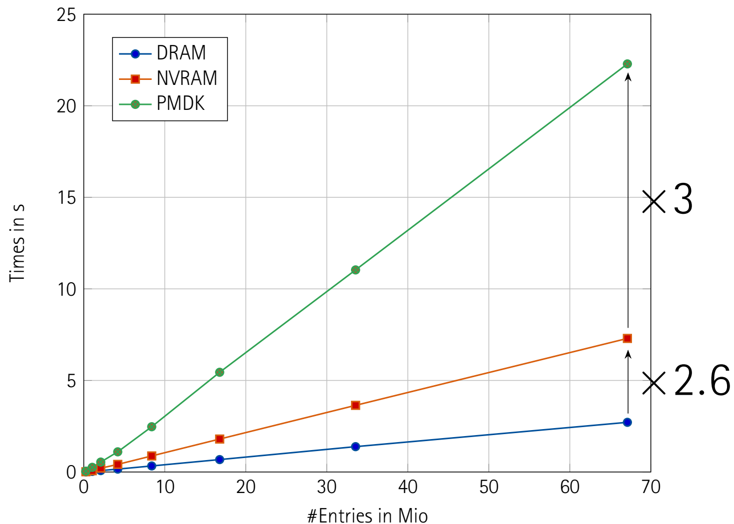
Linked List on DRAM, pure NVRAM and PMDK



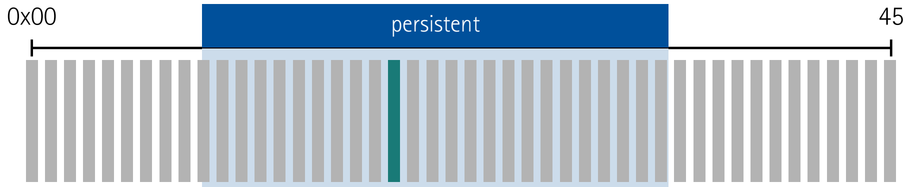
Linked List on DRAM, pure NVRAM and PMDK



Linked List on DRAM, pure NVRAM and PMDK



Problems in PMDK

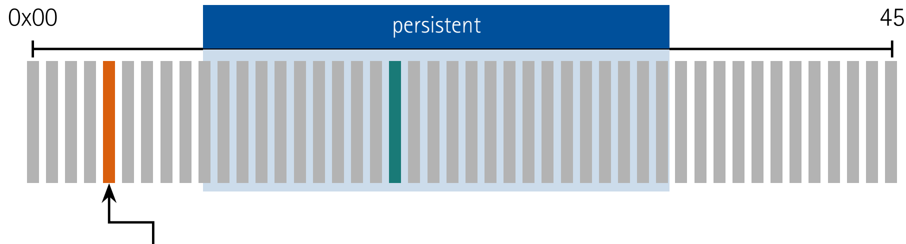


Pointer := (Pool-UUID, Offset)

Problems:

1. Fat-Pointer dereference incurs hefty indirection overhead

Problems in PMDK

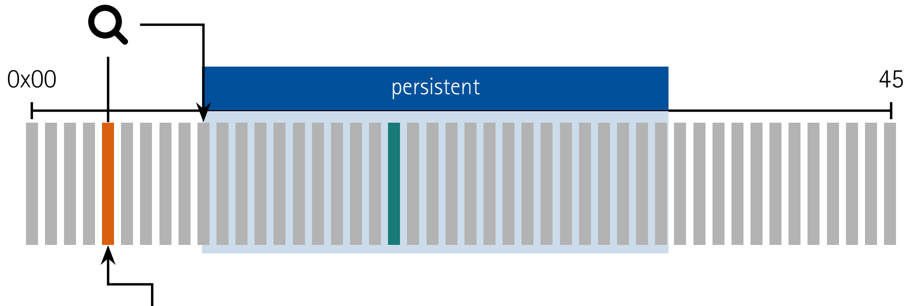


Pointer := (Pool-UUID, Offset)

Problems:

1. Fat-Pointer dereference incurs hefty indirection overhead

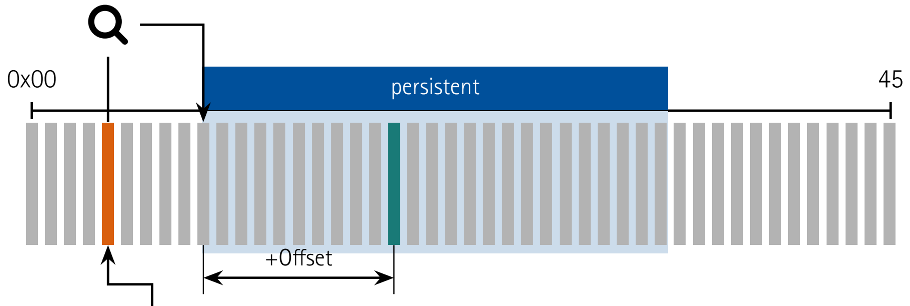
Problems in PMDK



Problems:

1. Fat-Pointer dereference incurs hefty indirection overhead

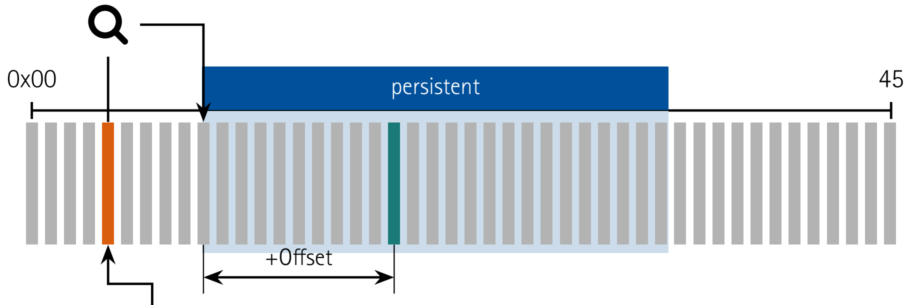
Problems in PMDK



Problems:

1. Fat-Pointer dereference incurs hefty indirection overhead

Problems in PMDK



Pointer := (Pool-UUID, Offset)

Problems:

1. Fat-Pointer dereference incurs hefty indirection overhead
2. Transactions and crash consistency

ISE SRA

Leibniz Universität Hannover

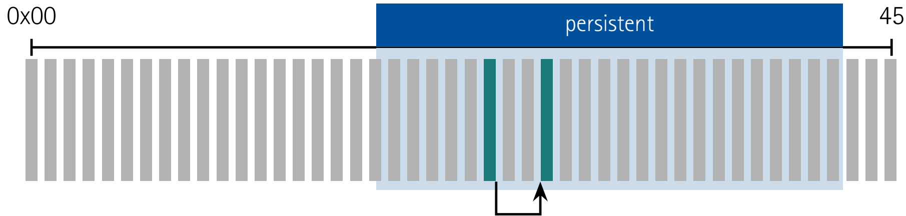
Our Approach

Our Approach

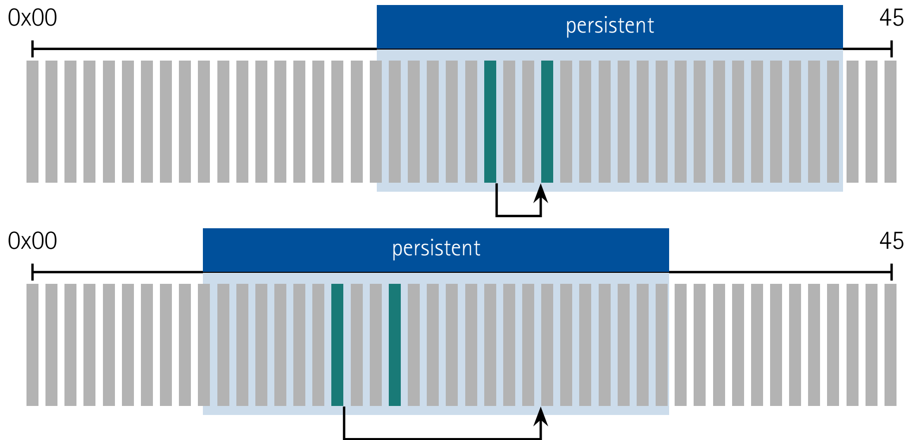
- Reduce indirections and overheads
  - Persist location as well as value
  - Don't inflate pointers

Volatile Address Spaces

Volatile Address Spaces



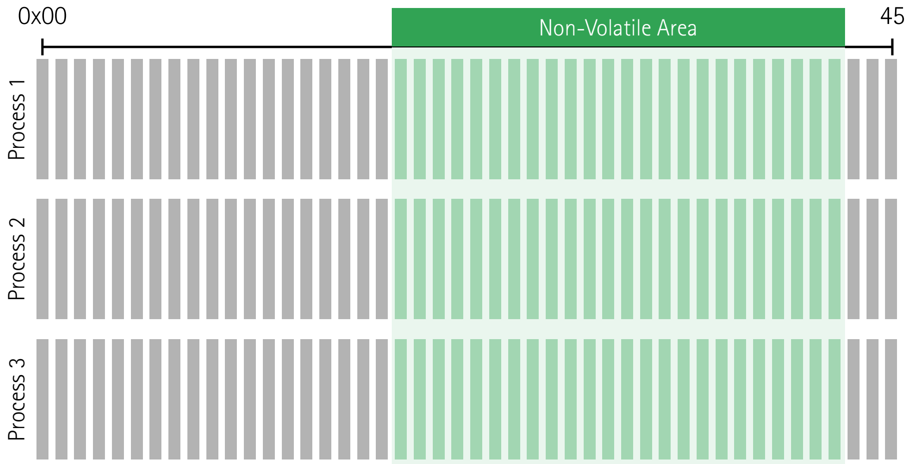
Volatile Address Spaces



Non-Volatile Address Spaces

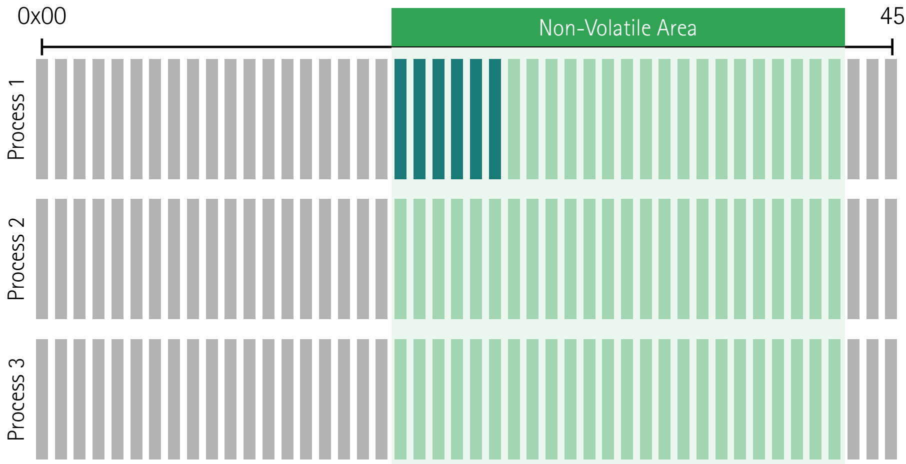
Non-Volatile Address Spaces

- Block an area of virtual memory for mapping NVRAM in every process



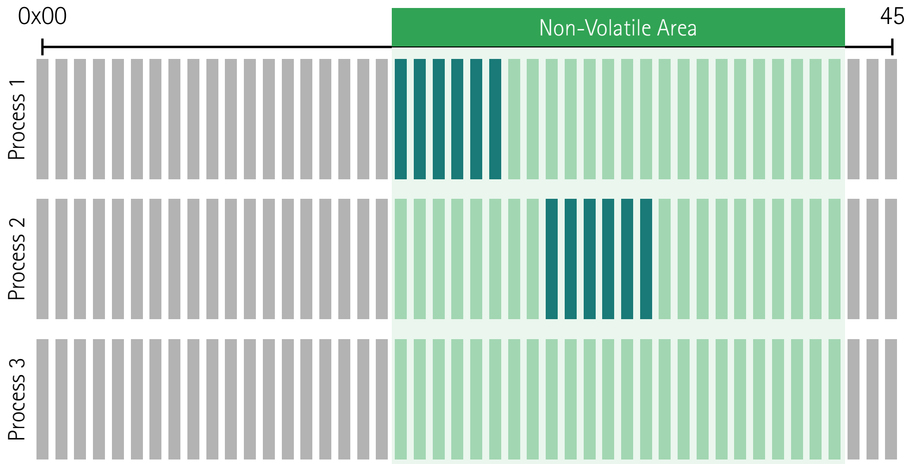
Non-Volatile Address Spaces

- Block an area of virtual memory for mapping NVRAM in every process
- **`mmap`** maps the device to the same address every time



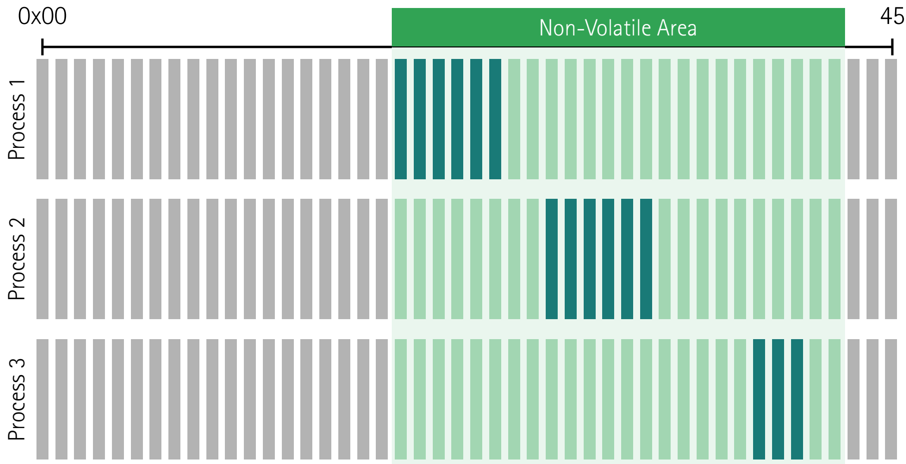
Non-Volatile Address Spaces

- Block an area of virtual memory for mapping NVRAM in every process
- **`mmap`** maps the device to the same address every time



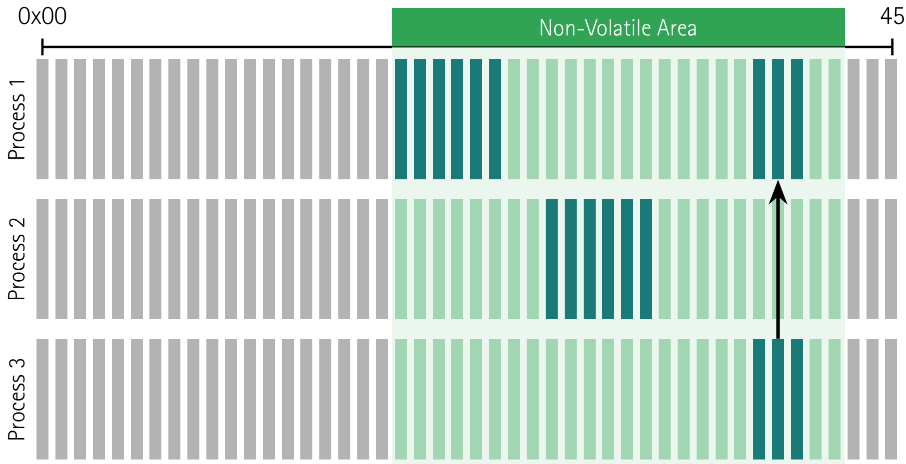
Non-Volatile Address Spaces

- Block an area of virtual memory for mapping NVRAM in every process
- **`mmap`** maps the device to the same address every time



Non-Volatile Address Spaces

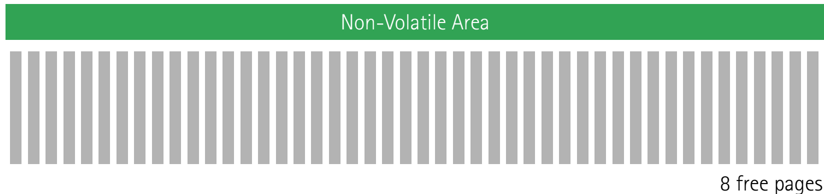
- Block an area of virtual memory for mapping NVRAM in every process
- **mmap** maps the device to the same address every time



Worst-Case: **sizeof(**Non-Volatile Area**)**

Assumptions

- New NVRAM namespaces are mapped first-fit inside our Non-Volatile area
- Page-granularity map and unmap-operations
- Discontinuous physical chunks are mapped continuously

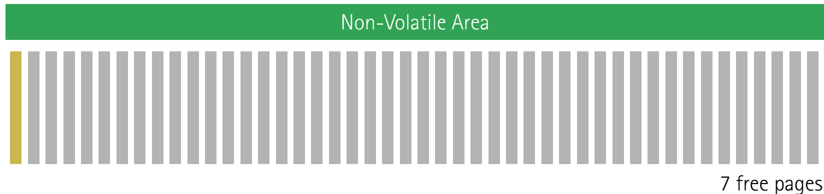


needed space $= 0$

Worst-Case: **sizeof(**Non-Volatile Area**)**

Assumptions

- New NVRAM namespaces are mapped first-fit inside our Non-Volatile area
- Page-granularity map and unmap-operations
- Discontinuous physical chunks are mapped continuously

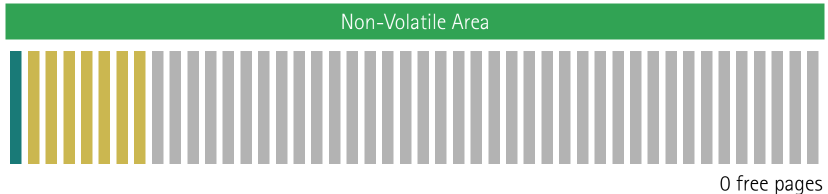


needed space $= 1$

Worst-Case: **sizeof(**Non-Volatile Area**)**

Assumptions

- New NVRAM namespaces are mapped first-fit inside our Non-Volatile area
- Page-granularity map and unmap-operations
- Discontinuous physical chunks are mapped continuously

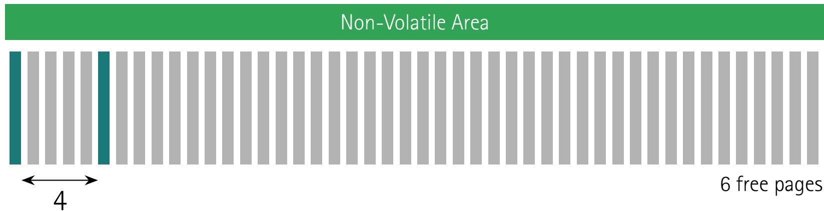


$$\text{needed space} = 8$$

Worst-Case: **sizeof(**Non-Volatile Area**)**

Assumptions

- New NVRAM namespaces are mapped first-fit inside our Non-Volatile area
- Page-granularity map and unmap-operations
- Discontinuous physical chunks are mapped continuously

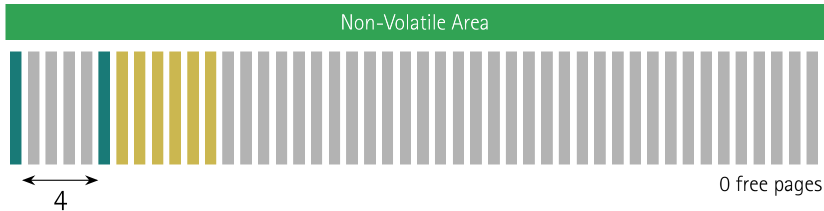


$$\text{needed space} = 4 + 2$$

Worst-Case: **sizeof(**Non-Volatile Area**)**

Assumptions

- New NVRAM namespaces are mapped first-fit inside our Non-Volatile area
- Page-granularity map and unmap-operations
- Discontinuous physical chunks are mapped continuously

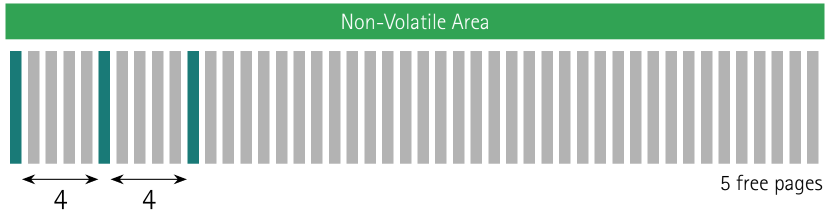


$$\text{needed space} = 4 + 2 + 6$$

Worst-Case: **sizeof(**Non-Volatile Area**)**

Assumptions

- New NVRAM namespaces are mapped first-fit inside our Non-Volatile area
- Page-granularity map and unmap-operations
- Discontinuous physical chunks are mapped continuously

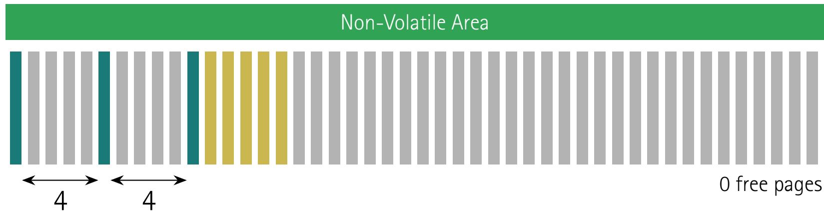


$$\text{needed space} = 2 \times 4 + 3$$

Worst-Case: **sizeof(**Non-Volatile Area**)**

Assumptions

- New NVRAM namespaces are mapped first-fit inside our Non-Volatile area
- Page-granularity map and unmap-operations
- Discontinuous physical chunks are mapped continuously

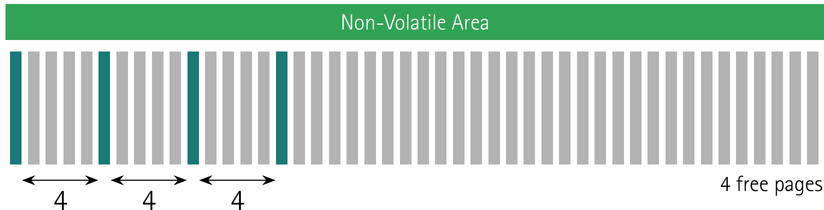


$$\text{needed space} = 2 \times 4 + 3 + 5$$

Worst-Case: **sizeof(**Non-Volatile Area**)**

Assumptions

- New NVRAM namespaces are mapped first-fit inside our Non-Volatile area
- Page-granularity map and unmap-operations
- Discontinuous physical chunks are mapped continuously

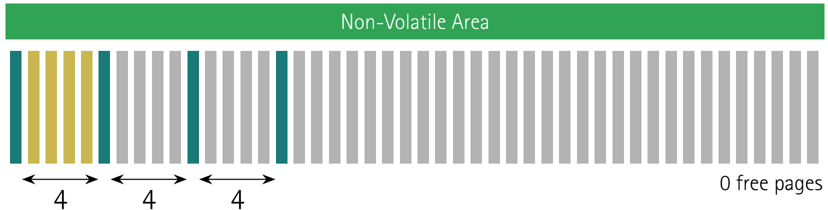


$$\text{needed space} = 3 \times 4 + 4$$

Worst-Case: **sizeof(**Non-Volatile Area**)**

Assumptions

- New NVRAM namespaces are mapped first-fit inside our Non-Volatile area
- Page-granularity map and unmap-operations
- Discontinuous physical chunks are mapped continuously

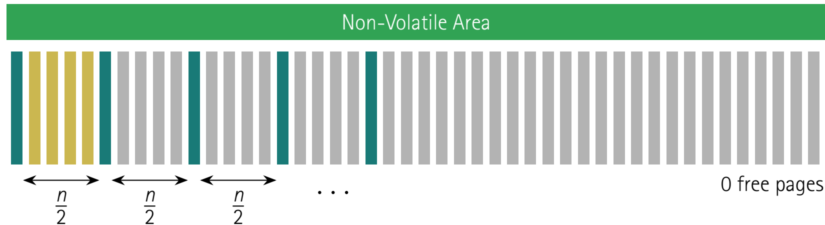


$$\text{needed space} = 3 \times 4 + 4$$

Worst-Case: **sizeof(**Non-Volatile Area**)**

Assumptions

- New NVRAM namespaces are mapped first-fit inside our Non-Volatile area
- Page-granularity map and unmap-operations
- Discontinuous physical chunks are mapped continuously

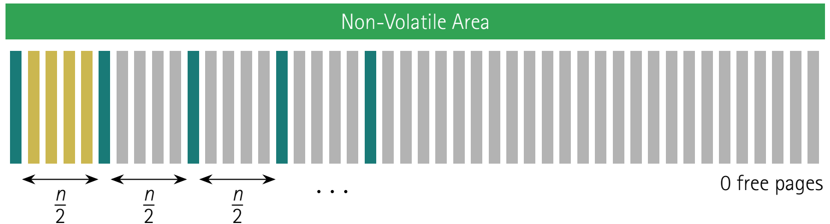


$$\text{needed space} = \frac{n}{2} \cdot \frac{n}{2} = \frac{1}{4} n^2$$

Worst-Case: **sizeof(**Non-Volatile Area**)**

Assumptions

- New NVRAM namespaces are mapped first-fit inside our Non-Volatile area
- Page-granularity map and unmap-operations
- Discontinuous physical chunks are mapped continuously



$$\text{needed space} = \frac{n}{2} \cdot \frac{n}{2} = \frac{1}{4} n^2$$

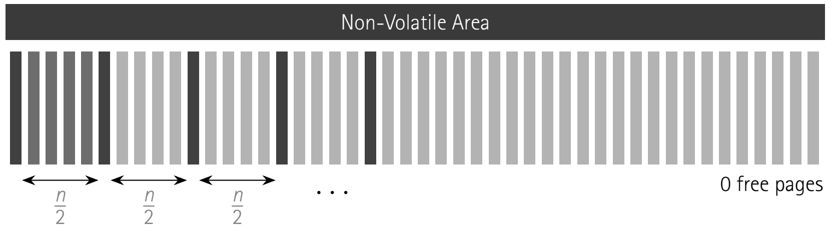
4 KiB pages: 2^{66} addresses

4 MiB pages: 2^{56} addresses

Worst-Case: **sizeof(**Non-Volatile Area**)**

Assumptions

- New NVRAM namespaces are mapped first-fit inside our Non-Volatile area
- ~~Page-granularity map and unmap-operations~~
- ~~Discontinuous physical chunks are mapped continuously~~



$$\text{needed space} = \frac{n}{2} \cdot \frac{n}{2} = \frac{1}{4} n^2$$

4 KiB pages: 2^{66} addresses

4 MiB pages: 2^{56} addresses

Our Interfaces

- **Persistent locations of objects in NVRAM**
  Allows using normal pointers and brings compatibility with existing libraries
- **Memory allocations with a modified musl malloc**
  Allocate a persistent heap
- **Management of root-pointers**
  Store and find root-objects, used for referencing all other objects in NVRAM

Case Study: XML-Parsing Done Once

- Modified libxml2 using a peristent heap
- Parse the XML document to DOM-representation ("cached" for later use)

Case Study: XML-Parsing Done Once

- Modified libxml2 using a peristent heap
- Parse the XML document to DOM-representation (“cached” for later use)
- First experiments:
  - First call: Parse a 1.5 MB RSS file into DOM-representation
  - Traverse the DOM and output HTML-list
  - Second call: Use the cached DOM-tree

Case Study: XML-Parsing Done Once

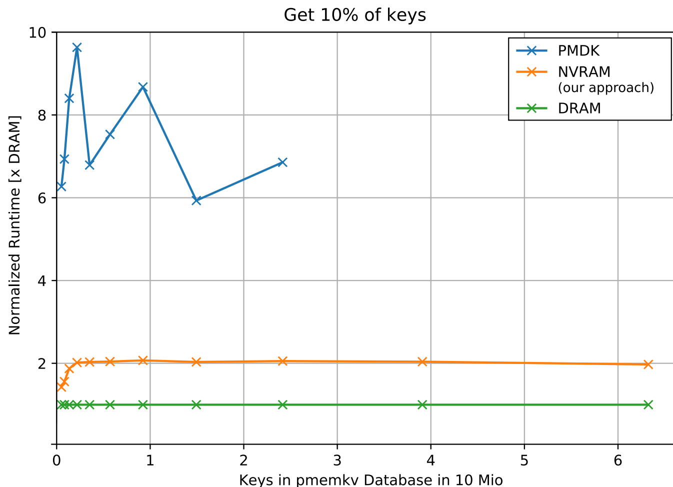
- Modified libxml2 using a peristent heap
- Parse the XML document to DOM-representation (“cached” for later use)
- First experiments:
  - First call: Parse a 1.5 MB RSS file into DOM-representation
  - Traverse the DOM and output HTML-list
  - Second call: Use the cached DOM-tree
  - Preliminary results show:
    - DRAM-Operation takes factor 4× longer than parsing once, then using the cache
    - With NVRAM-parsing takes double the time of using the cached DOM

Case Study: XML-Parsing Done Once

- Modified libxml2 using a peristent heap
- Parse the XML document to DOM-representation ("cached" for later use)
- First experiments:
  - First call: Parse a 1.5 MB RSS file into DOM-representation
  - Traverse the DOM and output HTML-list
  - Second call: Use the cached DOM-tree
  - Preliminary results show:
    - DRAM-Operation takes factor 4× longer than parsing once, then using the cache
    - With NVRAM-parsing takes double the time of using the cached DOM
- Best-Effort Consistency is good enough – if it crashes, parse the original XM

Case Study: Key-Value Store PMEMKV

- Based on Concurrent Hash Map from Intels Threading Building Blocks
- PMDK failed with Out-Of-Memory with 11 GB NVRAM at 39 Mio Entries



Conclusion

- PMDK tries to solve everything
  - Performance overheads
  - Memory Overheads

Persistence with
File Systems

Huge DRAM

Huge Persistent
Best-Effort Cache

Conclusion

- PMDK tries to solve everything
  - Performance overheads
  - Memory Overheads
- We modified Linux to make the location of NVRAM-objects persistent
  - Allows usage of native pointers
  - Adds compatibility for existing libraries

Persistence with File Systems

Huge DRAM

Huge Persistent Best-Effort Cache

Conclusion

- PMDK tries to solve everything
  - Performance overheads
  - Memory Overheads
- We modified Linux to make the location of NVRAM-objects persistent
  - Allows usage of native pointers
  - Adds compatibility for existing libraries
- No crash consistency $\rightarrow$ future work

Persistence with File Systems

Huge DRAM

Huge Persistent Best-Effort Cache

Conclusion

- PMDK tries to solve everything
  - Performance overheads
  - Memory Overheads
- We modified Linux to make the location of NVRAM-objects persistent
  - Allows usage of native pointers
  - Adds compatibility for existing libraries
- No crash consistency $\rightarrow$ future work

Persistence with File Systems

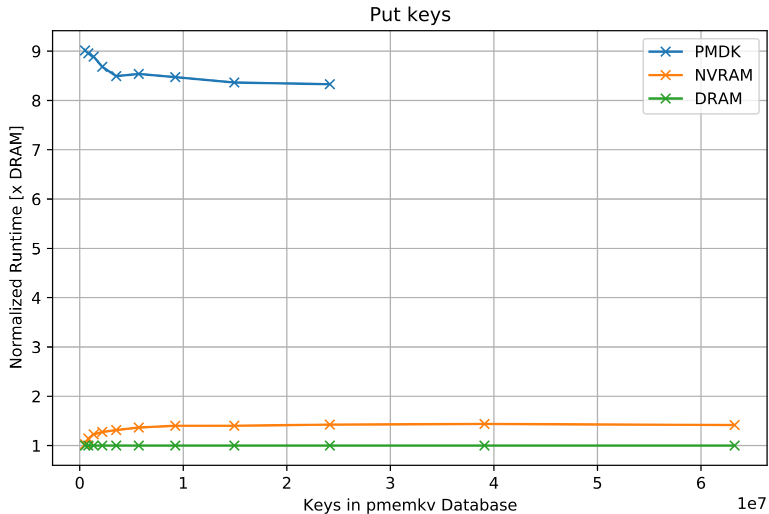
Huge DRAM

Huge Persistent Best-Effort Cache

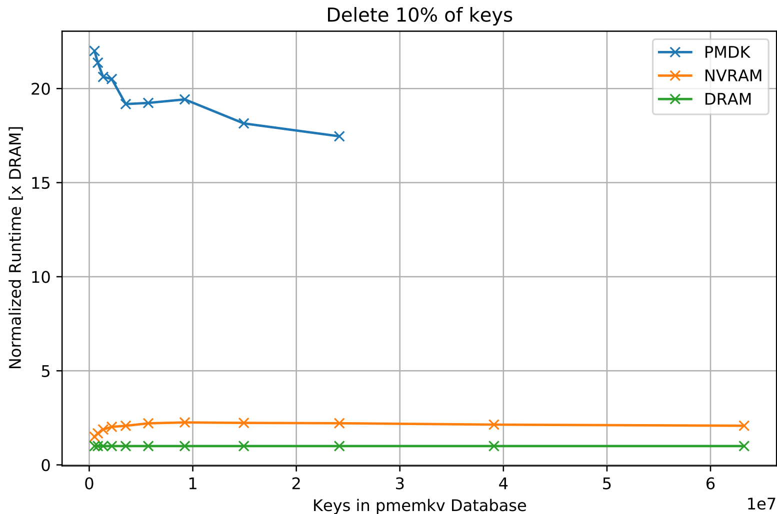
Stefan Naumann

Email: naumann@sra.uni-hannover.de

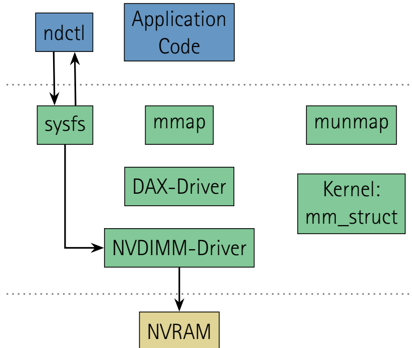
PMEMKV: Put



PMEMKV: Delete

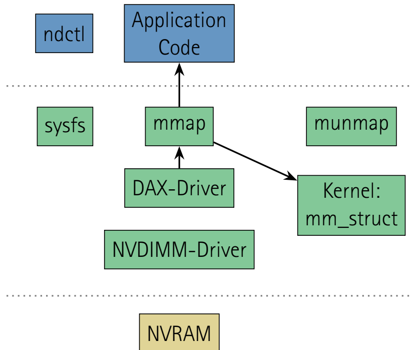


Lifecycle of NVRAM device



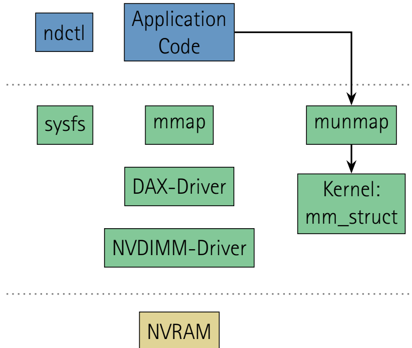
- Create Namespace - destination pointer from user to driver

Lifecycle of NVRAM device



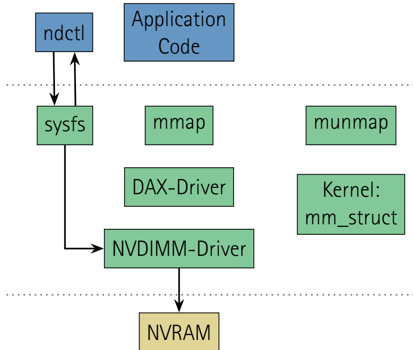
- Create Namespace – destination pointer from user to driver
- `mmap` the device into user space

Lifecycle of NVRAM device



- Create Namespace – destination pointer from user to driver
- `mmap` the device into user space
- `munmap` the device

Lifecycle of NVRAM device



- Create Namespace - destination pointer from user to driver
- `mmap` the device into user space
- `munmap` the device
- Destroy Namespace
  free partition for later re-use