Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

nvm_malloc: Memory Allocation for NVRAM

David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, Hasso Plattner

nvm_malloc

- General-purpose allocator concept for NVRAM
- Performant, failure-atomic allocation
- Easy and fast recovery through named allocations
- User-mode library, usable by any application
- https://github.com/hyrise/nvm_malloc

Current Memory Technologies

fast

direct access

OR

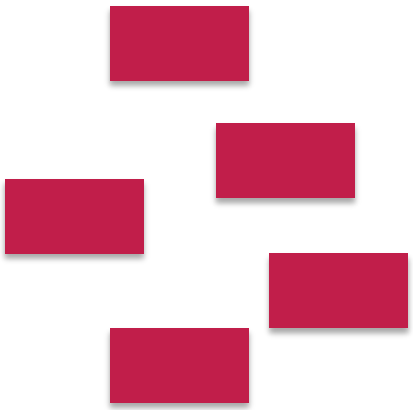
non-volatile

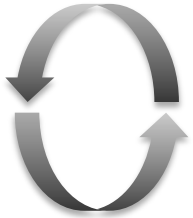
high capacity

cheap

Persistency == Redundancy

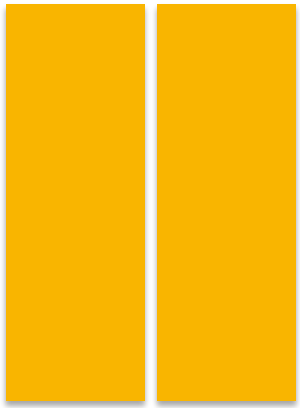
volatile





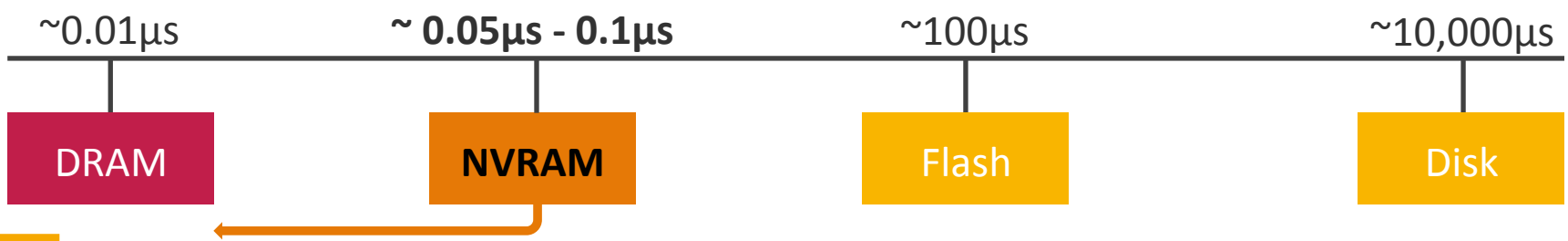
Transformation required

non-volatile

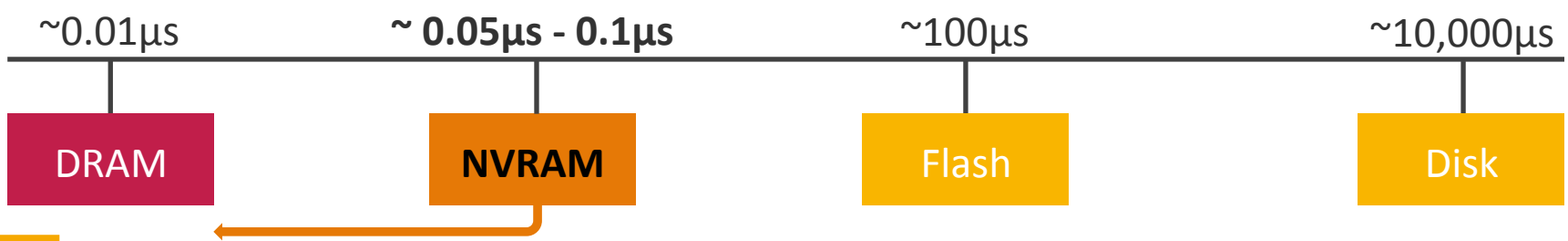


NVRAM

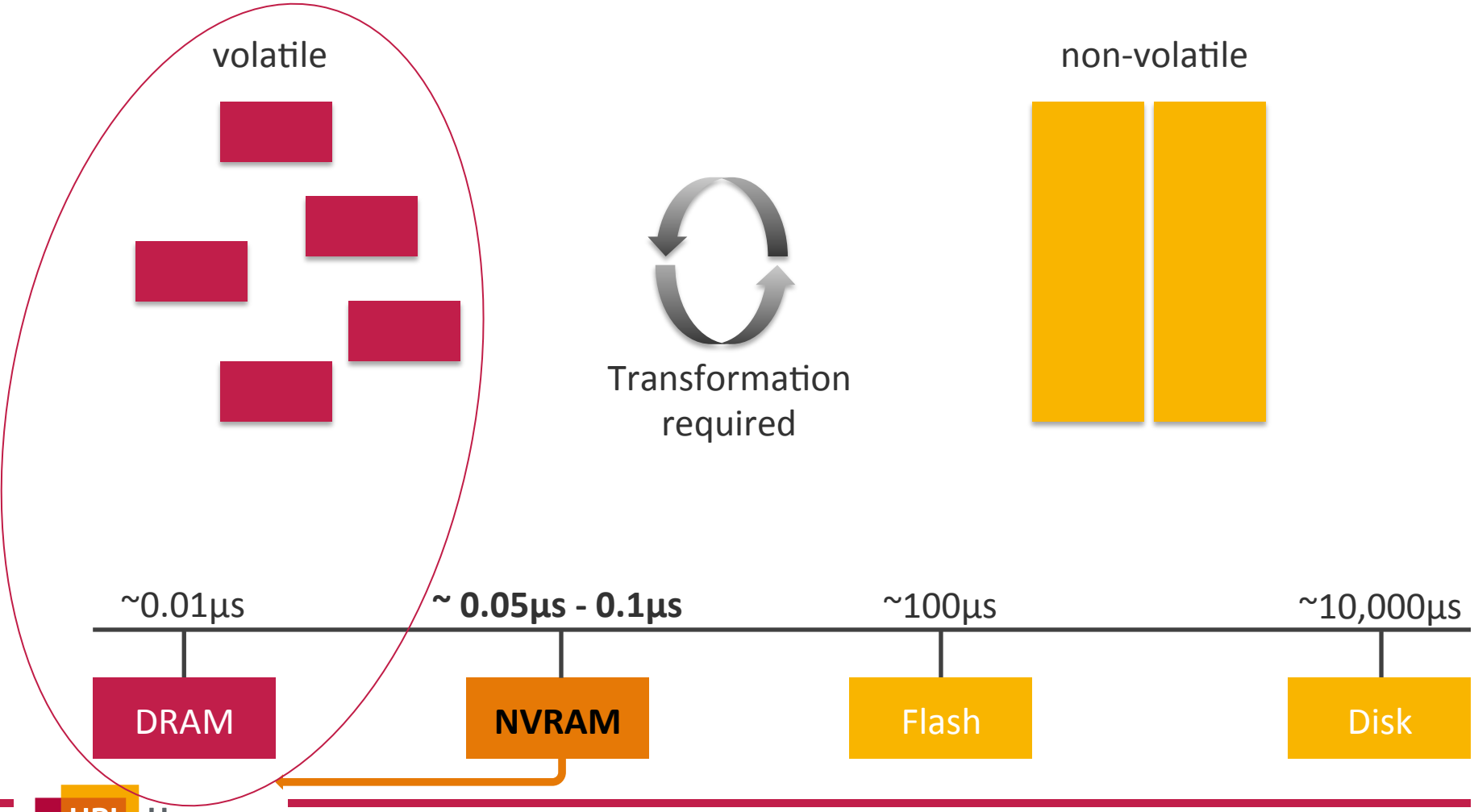
- non-volatile byte-adressable memory
- faster and more durable than flash
- higher capacity, cheaper, and less energy consumption than DRAM



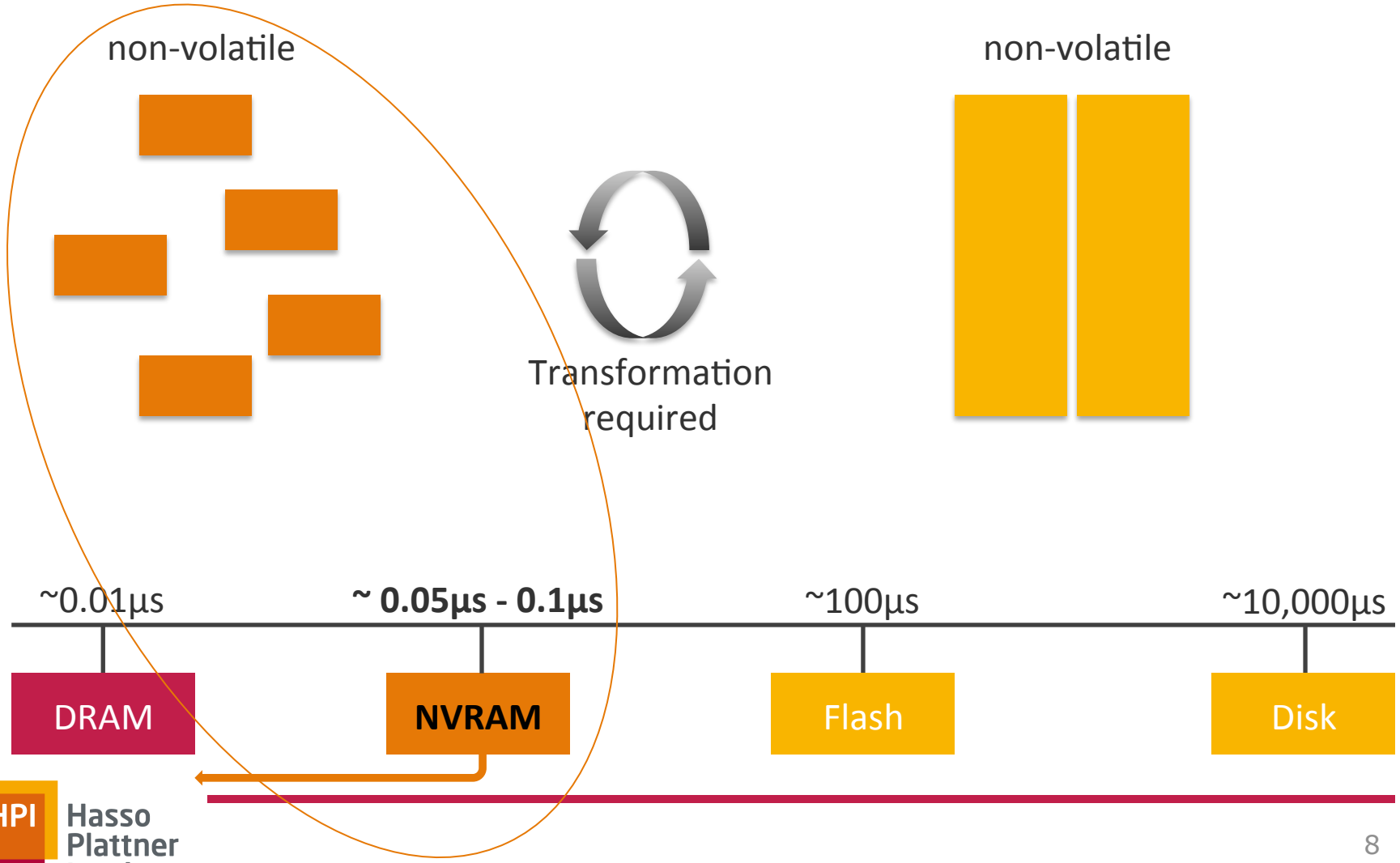
Persistency is redundant



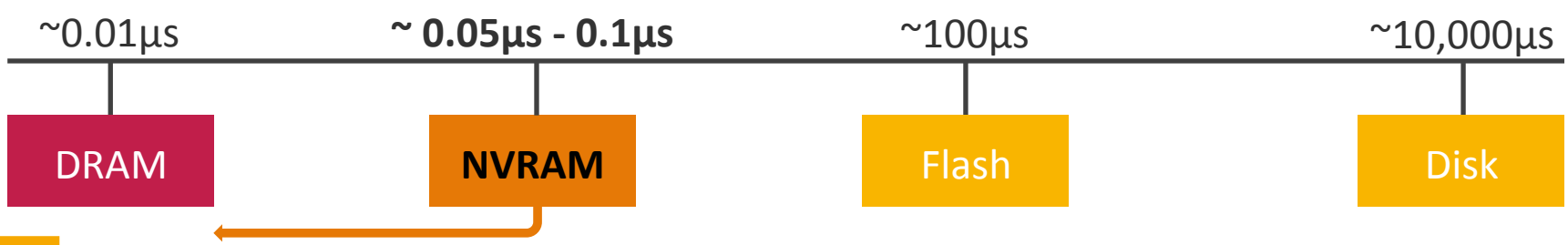
Persistency is redundant



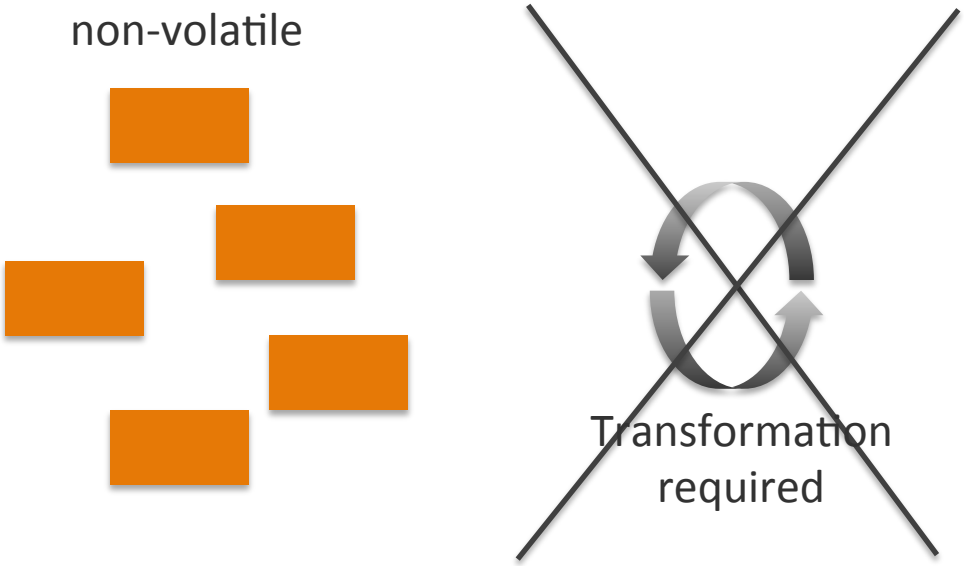
Persistency is redundant

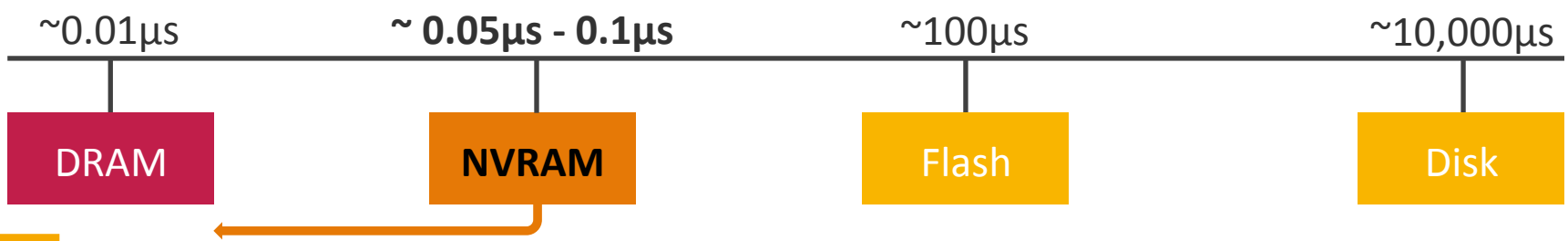


Persistency is redundant



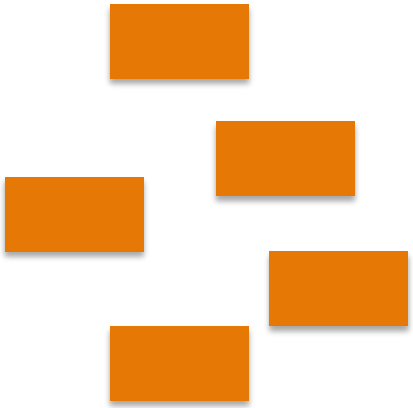
Persistency is redundant

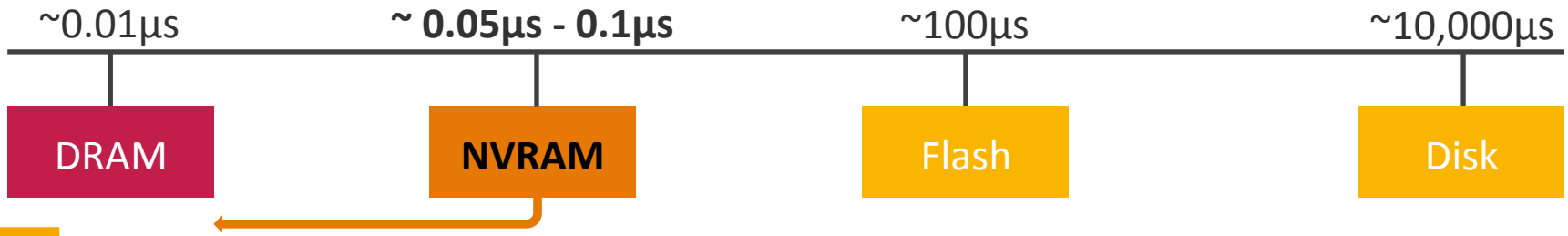




Persistency is redundant

non-volatile





Challenges for Data Structures

- Volatile CPU caches hold data that is not on NVRAM
- NVRAM may have inconsistent states if stores are reordered
  - Compiler-Level Reordering, CPU Out-Of-Order Execution
- Incomplete updates need to be rolled back
- NVRAM memory has to be dynamically managed
- Pointers to NVRAM objects need to be valid after a restart

Requirements for Allocators

- Volatile Allocators need to
  - deliver pointer to memory of requested size and free it
  - be performant
- Non-volatile Allocators also need to
  - rediscover previous allocations
  - ensure crashes do not create non-volatile memory leaks

Safe Allocation

- Imagine an insert into a linked list (between `prev_node` and `next_node`)

```
NODE* new_node = malloc(sizeof(NODE));
new_node->next = next_node;
prev_node->next = new_node;
```

Safe Allocation

- Imagine an insert into a linked list (between `prev_node` and `next_node`)

```
NODE* new_node = malloc(sizeof(NODE));
new_node->next = next_node;
```

~~`prev_node->next = new_node;`~~

- Memory allocated for `new_node` is lost

Two-Step Allocation

- First introduced by pmemalloc [1]

- **Reserve** memory, prepare data, **activate** memory
- Reserve: finds a region and returns pointer, memory will be recovered as "free"
- Activate: establish links from other objects, memory will be recovered as "in use"

[1] Rudoff, https://github.com/pmem/linux-examples/tree/master/libpmemalloc. 2013

Introducing nvm_malloc

- `void* nvm_initialize(working_directory, recover_flag);`

- creates or recovers allocation arena
- no size has to be specified – arena is dynamically resized

Introducing nvm_malloc

- `void* nvm_reserve(size);`
- `void  nvm_activate(ptr, lp1, tgt1, lp2, tgt2);`
- `void  nvm_free(ptr);`

Introducing nvm_malloc

- Insert into a linked list

```
NODE* new_node = nvm_reserve(sizeof(NODE));
new_node->next = next_node;
nvm_activate(new_node, &prev_node->next, new_node,
    NULL, NULL);
```

Introducing nvm_malloc

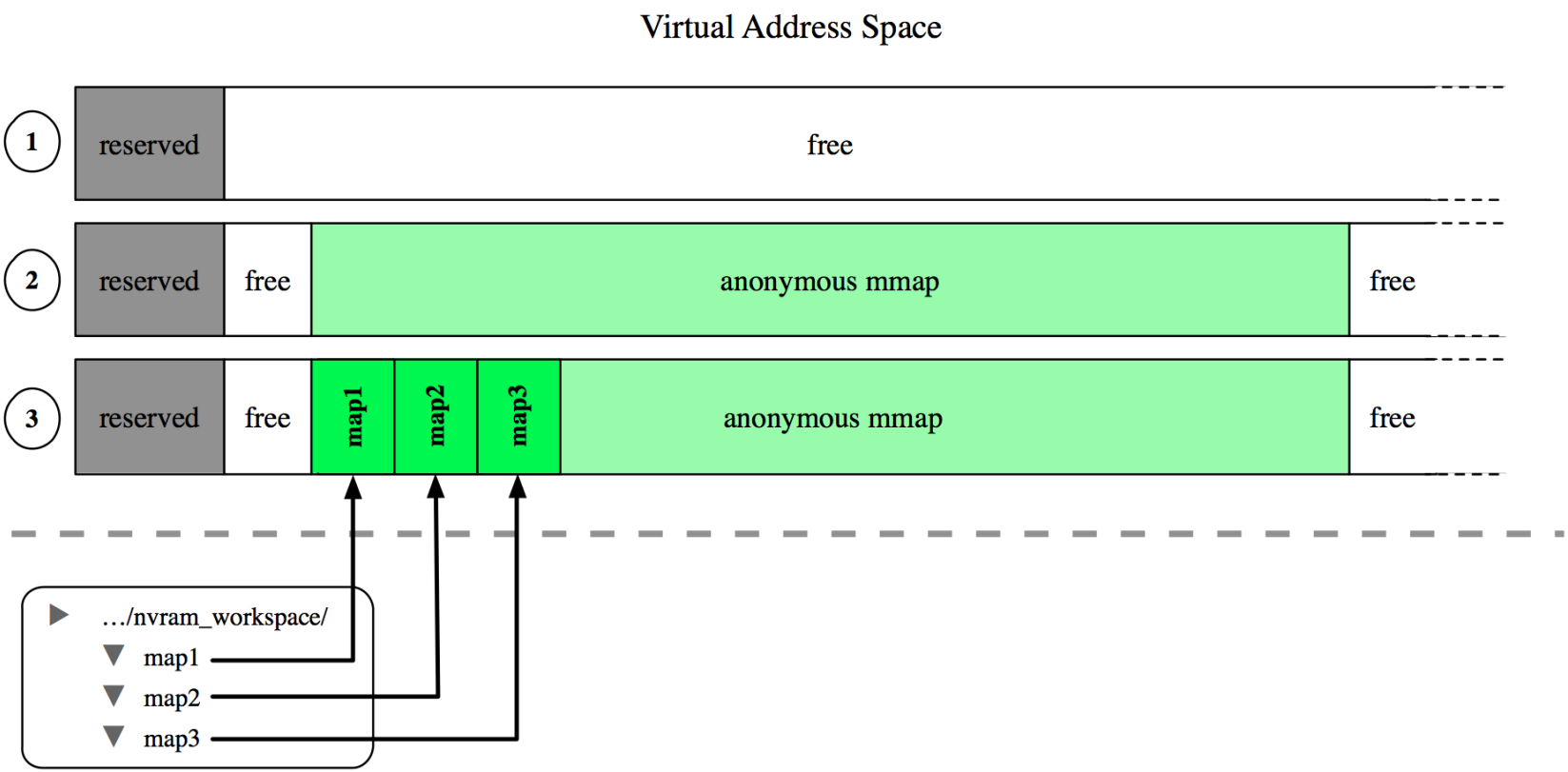
- How to retrieve previous allocations?
- `void* nvm_reserve_id(id, size);`
- `void  nvm_activate_id(id);`
- `void  nvm_get_id(id);`
- `void  nvm_free(id);`

Internals

- Memory is backed by files on a persistent filesystem (e.g., PMFS [2]) and `mmap`ped into the application's virtual space
- All translations are done by the MMU without kernel involvement or other software overhead
- Instead of creating one giant file, multiple files are mapped into contiguous, anonymous `mmap` space

[2] Dulloor et al., "System Software for Persistent Memory", EuroSys, 2014

Internals


Virtual Address Space
1
reserved
free
2
reserved
free
anonymous mmap
free
3
reserved
free
map1
map2
map3
anonymous mmap
free
…/nvram_workspace/
map1
map2
map3

Internals

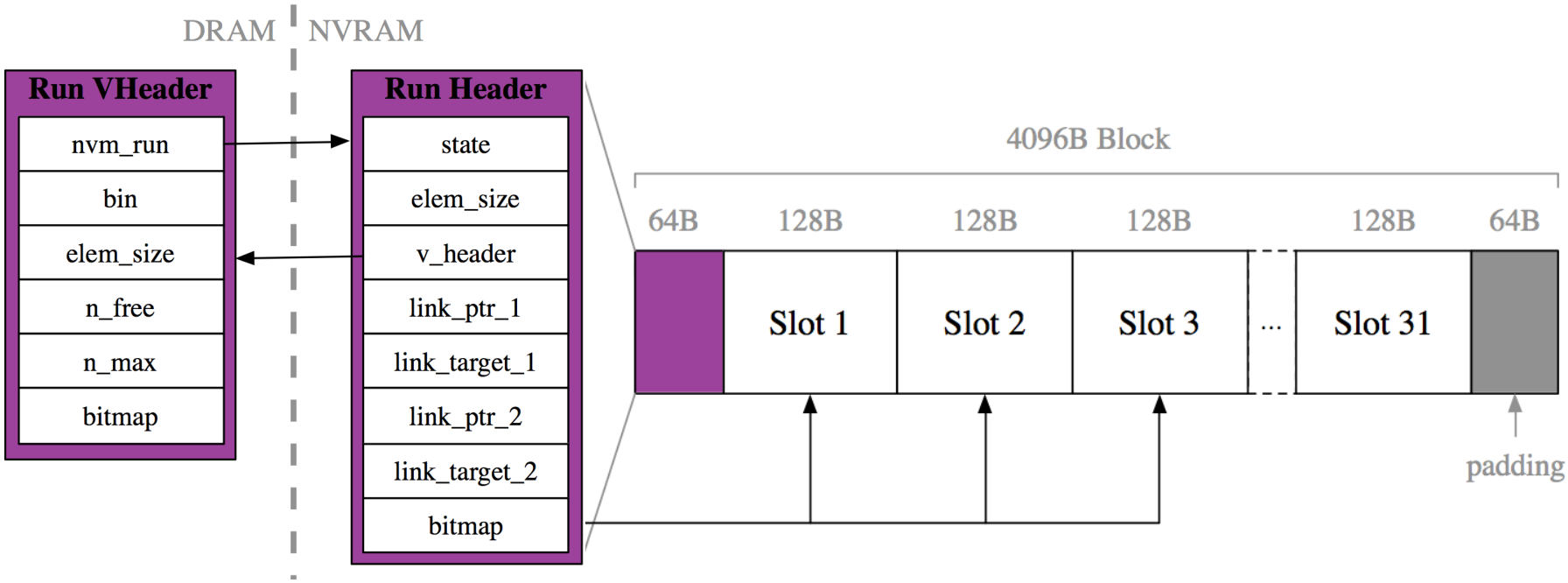
- Memory management is based on jemalloc [3]
- Backing files represent 4MB chunks
- Allocations are placed in three size categories: small (<2KB), large (2KB-2MB), huge (>2MB)
- Small allocations are grouped by size, large allocations use one or more blocks, huge allocations are stored directly in chunks

[3] J. Evans. “A scalable concurrent malloc (3) implementation for freebsd”. In Proc. of the BSDCan Conference, 2006.

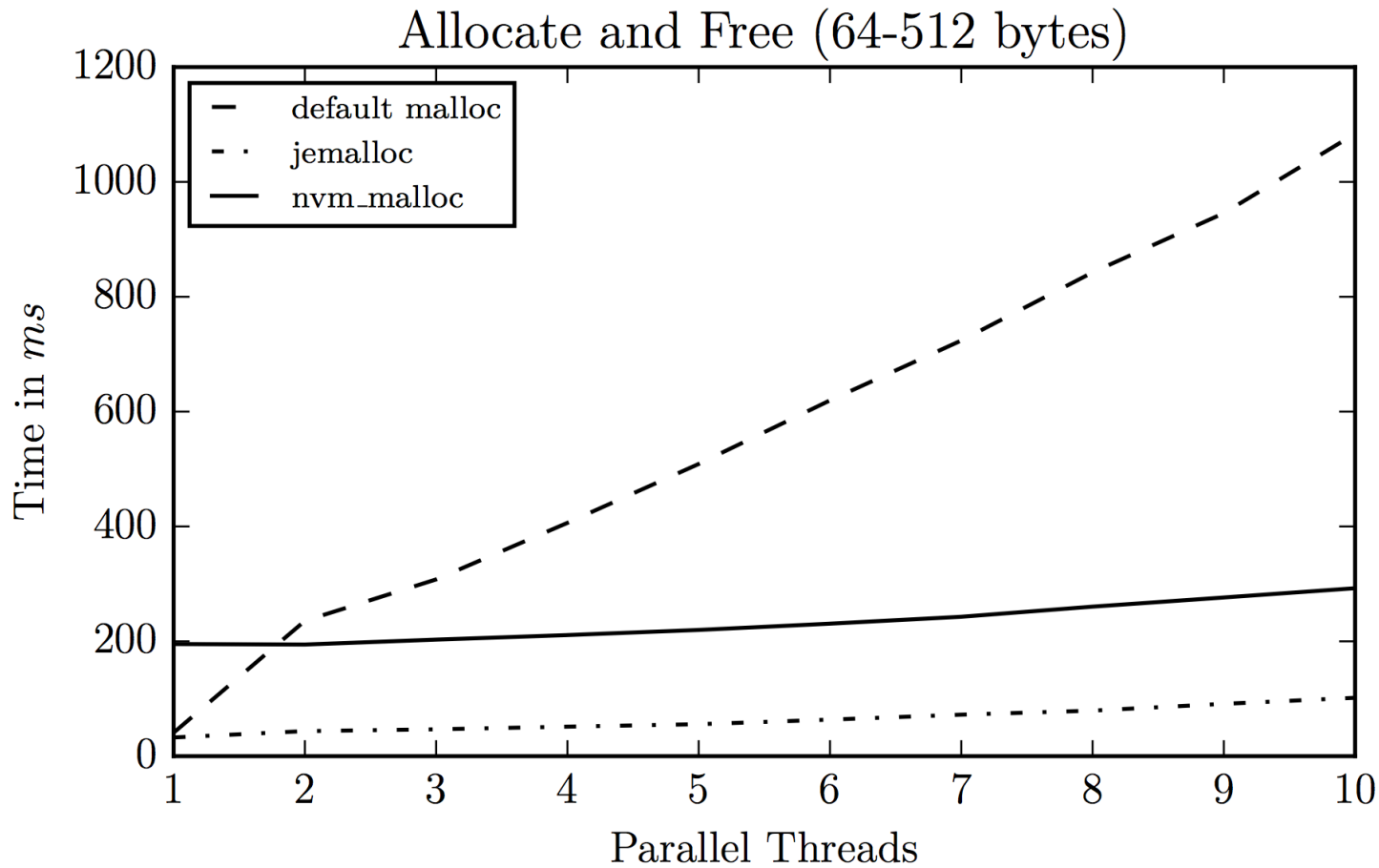
Internals

- Every chunk and every block has a header
- 64 B (cacheline-sized) headers ensure atomic flushes to NVRAM
- Flushing the headers to NVRAM (using CLFLUSH/CLWB) on every modification is expensive
- To reduce their number, a volatile copy is kept in DRAM
- Reserve step can be executed in DRAM

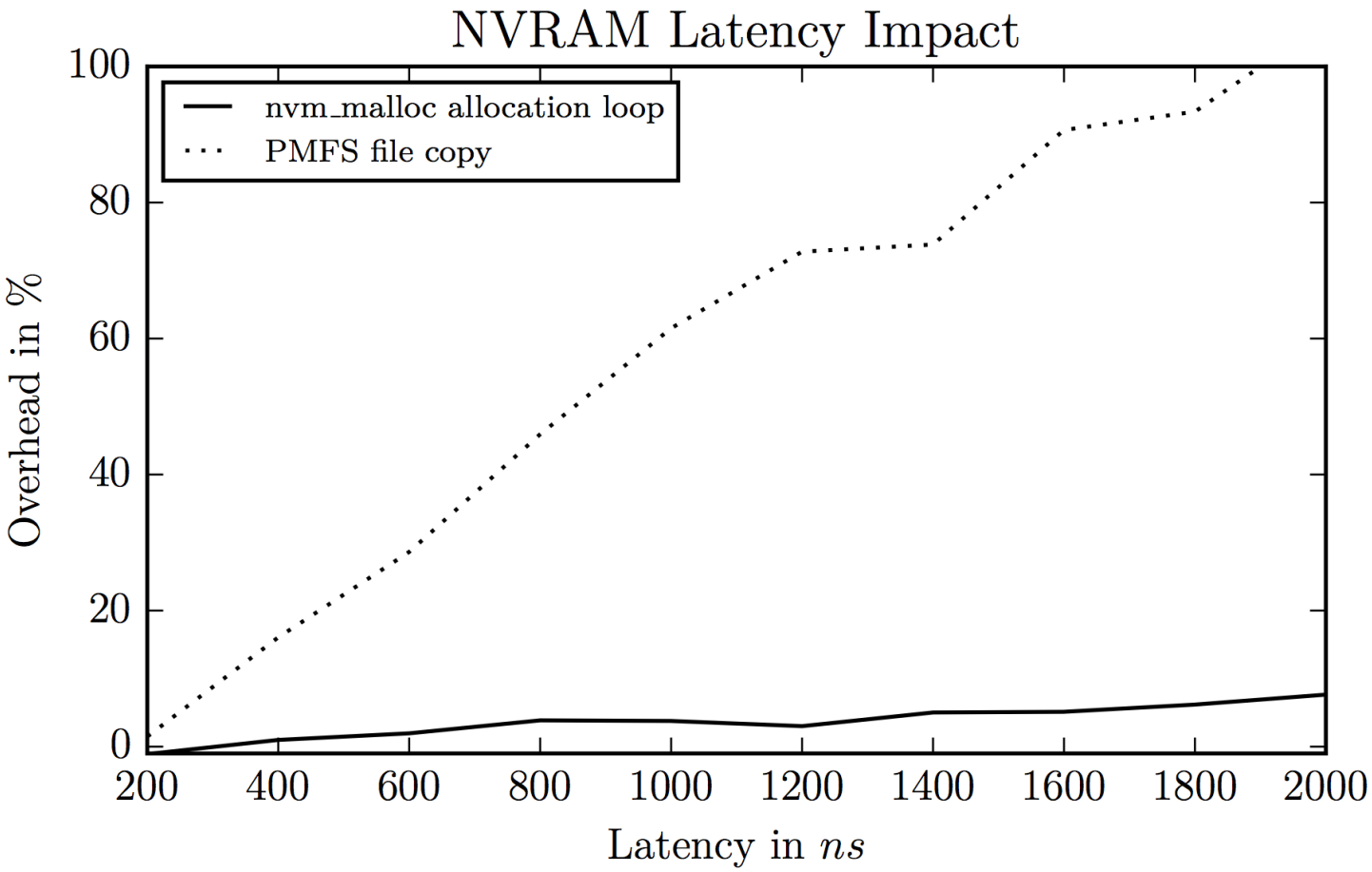
Internals


DRAM
NVRAM
Run VHeader
nvm_run
bin
elem_size
n_free
n_max
bitmap
Run Header
state
elem_size
v_header
link_ptr_1
link_target_1
link_ptr_2
link_target_2
bitmap
4096B Block
64B
128B
128B
128B
128B
64B
Slot 1
Slot 2
Slot 3
...
Slot 31
padding

Benchmarks


Allocate and Free (64-512 bytes)
default malloc
jemalloc
nvm_malloc
Time in ms
0
200
400
600
800
1000
1200
1
2
3
4
5
6
7
8
9
10
Parallel Threads

Benchmarks


NVRAM Latency Impact
nvm_malloc allocation loop
PMFS file copy
Overhead in %
0
20
40
60
80
100
200
400
600
800
1000
1200
1400
1600
1800
2000
Latency in ns

Related Work

- NVMalloc (Wang et al.): SSD-based, one mmap‘ped file per allocation
- NVMalloc (Moraru et al.): focus on wear-leveling, require CPU extension, not looking at recovery
- pmem.io: collection of persistent memory libraries, both low- and high-level

Summary

- nvm_malloc allows for atomic, consistent, and durable allocations on NVRAM
- two-step allocation prevent memory leaks
- virtual headers on DRAM improve performance
- It is available at https://github.com/hyrise/nvm_malloc

NVC-Hashmap: A Persistent and Concurrent Hashmap for Non-Volatile Memories
Today, 16:30 – 16:50 at IMDM

Backup

Other NVM allocators