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

- DRAM (~0.01µs)
- Flash (~100µs)
- Disk (~10,000µs)
Persistency == Redundancy

volatile

non-volatile

Transformation required

~0.01µs
DRAM

~100µs
Flash

~10,000µs
Disk
NVRAM

- non-volatile byte-addressable memory
- faster and more durable than flash
- higher capacity, cheaper, and less energy consumption than DRAM
Persistency is redundant

Transformation required

volatile

non-volatile

~0.01µs

~0.05µs - 0.1µs

~100µs

~10,000µs

DRAM

NVRAM

Flash

Disk
Persistency is redundant

Transformation required

volatile

non-volatile

~0.01µs

~ 0.05µs - 0.1µs

~100µs

~10,000µs

DRAM

NVRAM

Flash

Disk

HPI

Hasso Plattner Institut
Persistency is redundant

Transformation required

non-volatile

~0.01µs
DRAM

~0.05µs - 0.1µs
NVRAM

~100µs
Flash

~10,000µs
Disk

non-volatile
Persistency is redundant

Transformation required

non-volatile

~0.01µs

~ 0.05µs - 0.1µs

~100µs

~10,000µs

DRAM

NVRAM

Flash

Disk
Persistency is redundant

non-volatile

Transformation required

~0.01µs

~ 0.05µs - 0.1µs

~100µs

~10,000µs

DRAM

NVRAM

Flash

Disk
Persistency is redundant

non-volatile

DRAM

~0.01µs

NVRAM

~ 0.05µs - 0.1µs

Flash

~100µs

Disk

~10,000µs
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`)

```c
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”

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

```c
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 mapped 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

Internals

Virtual Address Space

1. reserved

2. reserved  free  anonymous mmap

3. reserved  free  map1  map2  map3  anonymous mmap

.../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

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

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

Parallel Threads

Parallel Threads
Benchmarks

![NVRAM Latency Impact Graph]

- nvm_malloc allocation loop
- PMFS file copy

Overhead in % vs. 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
Backup
Other NVM allocators

Linked List Benchmark Single Threaded

Time in ms

glibc  jemalloc  nvm_malloc  NVMalloc  pmemalloc

30  25  20  15  10  5  0

31