NVAll

A Crash-Resistant and Kernel-Compatible Memory Allocator for NVRAM

September 29, 2023

Dustin Nguyen[1], Ole Wiedemann[1],
Jörg Nolte[2], Wolfgang Schröder-Preikschat[1]

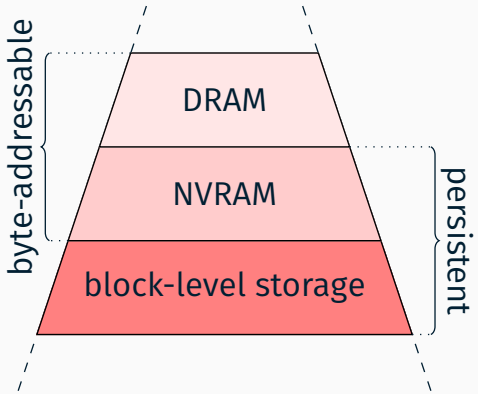
[1]Friedrich-Alexander-Universität Erlangen-Nürnberg,
[2]Brandenburgische Technische Universität Cottbus-Senftenberg

Funded by
DFG Deutsche Forschungsgemeinschaft
German Research Foundation
Project No.
501993201

Persistent Memory

Characteristics of Intel® Optane™ Persistent Memory

- byte-addressable
- persistent, no refresh required
- available in TiB
- faster than Flash, slower than DRAM
- better price/capacity ratio than DRAM



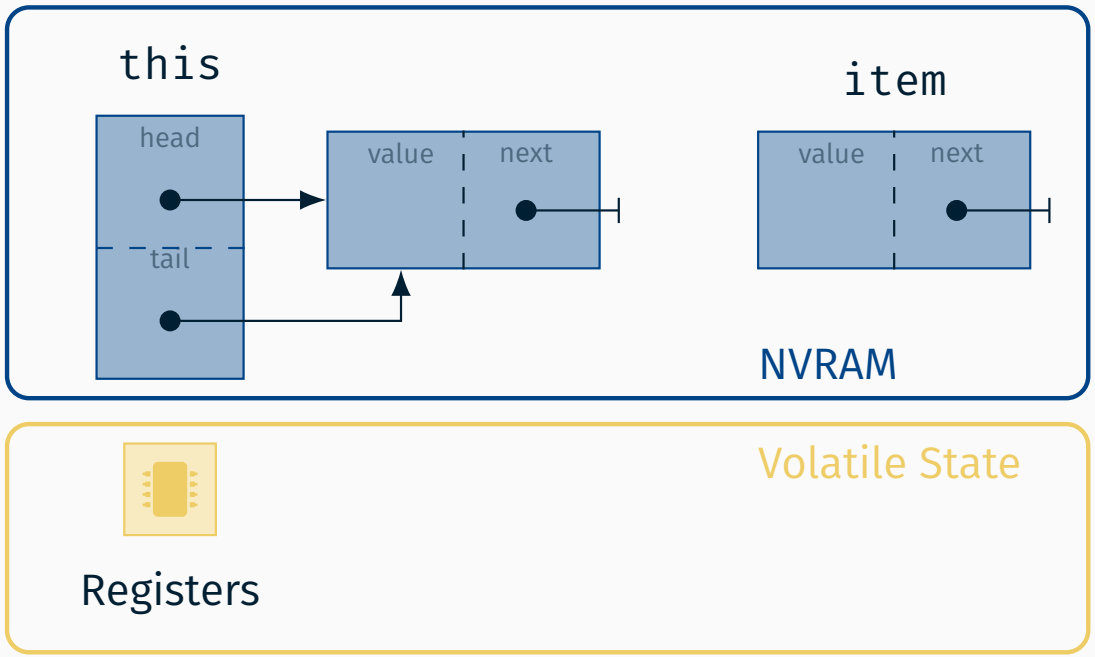
Non-Volatile Memory Challenges

```
void
enqueue(queue_t *this, chain_t *item)
{

    item->next=NULL;
    this->tail->next=item;

    this->tail=item;


}
```



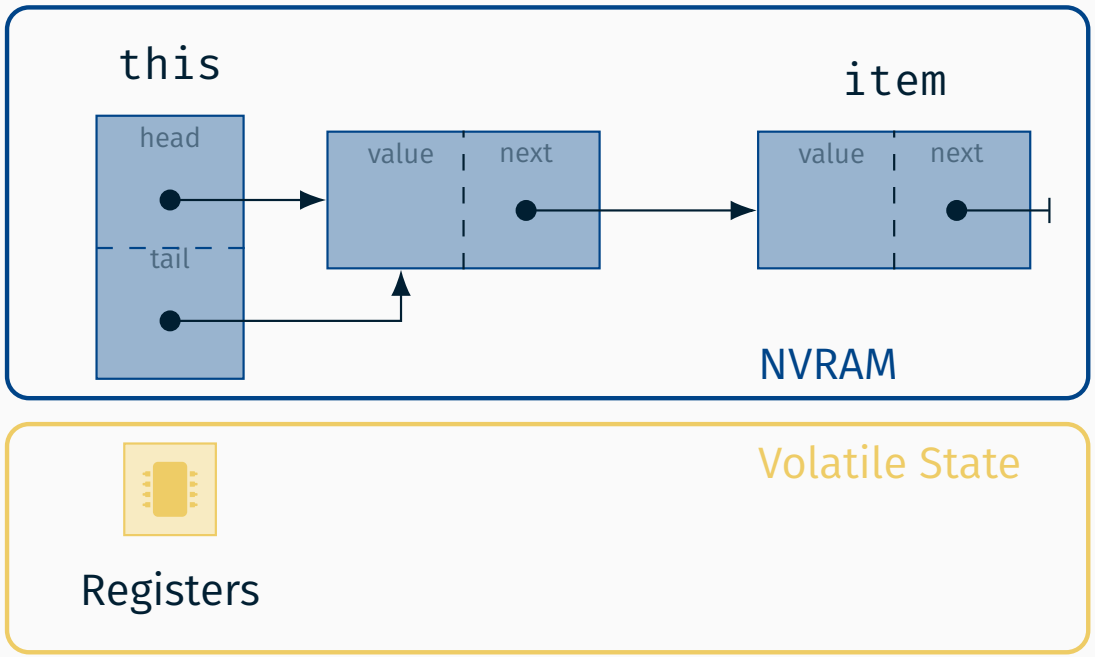
Non-Volatile Memory Challenges

```
void
enqueue(queue_t *this, chain_t *item)
{


    item->next=NULL;
    this->tail->next=item;

    this->tail=item;


}
```

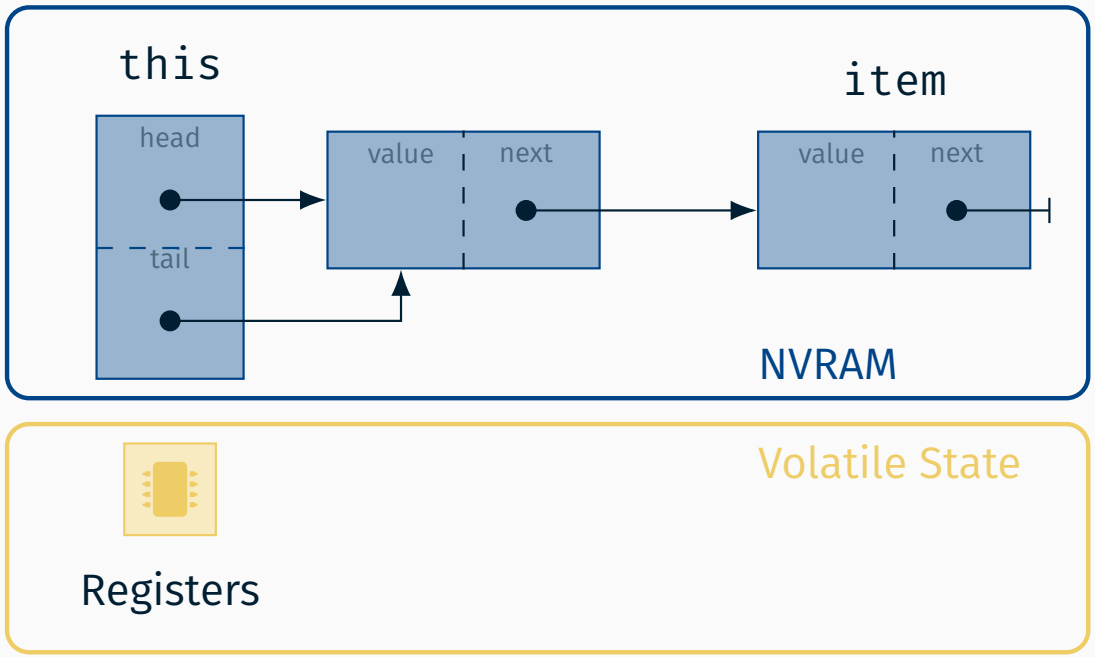


Non-Volatile Memory Challenges

```
void
enqueue(queue_t *this, chain_t *item)
{


    item->next=NULL;
    this->tail->next=item;
- - - - - - - - - - - - - - - - - -
    this->tail=item;


}
```

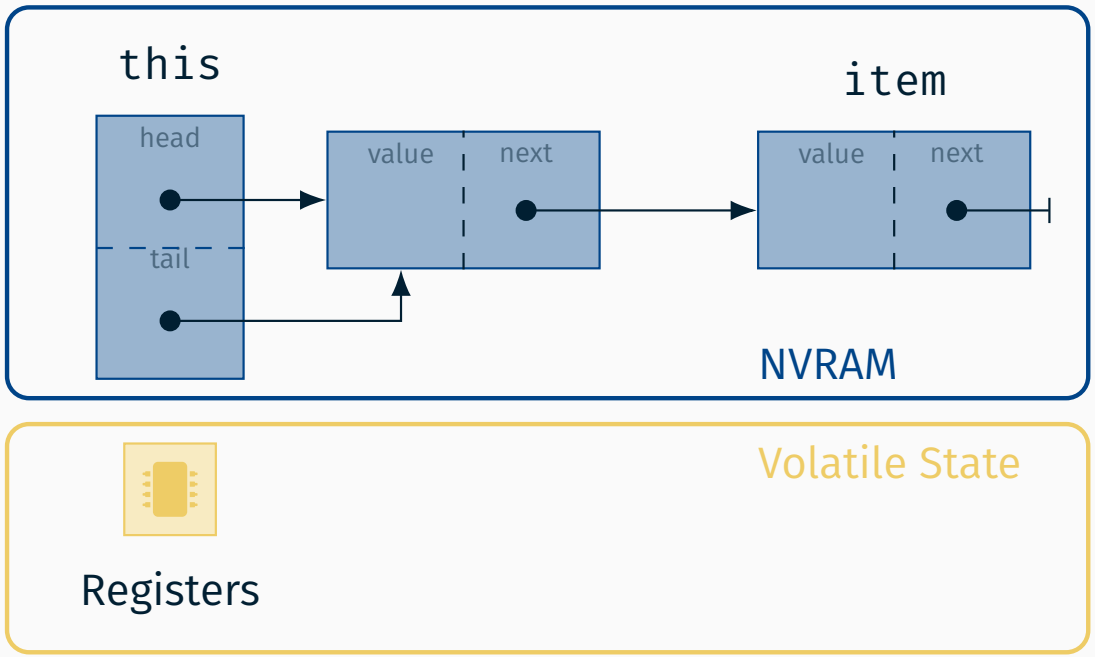


Non-Volatile Memory Challenges

```
void
enqueue(queue_t *this, chain_t *item)
{
    pthread_mutex_lock(&this->m);

    item->next=NULL;
    this->tail->next=item;
- - - - - - - - - - - - - - - - - - -
    this->tail=item;

    pthread_mutex_unlock(&this->m);
}
```



Non-Volatile Memory Challenges

```
void
enqueue(queue_t *this, chain_t *item)
{
    pthread_mutex_lock(&this->m);

    item->next=NULL;
    this->tail->next=item;
- - - - - - - - - - - - - - - - - -
    this->tail=item;

    pthread_mutex_unlock(&this->m);
}
```

Broken Time Machine

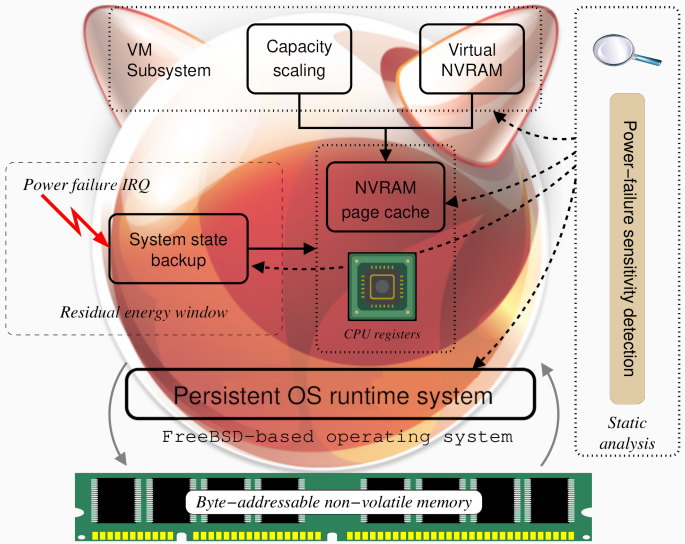
- inconsistent `tail`-pointer
- mutexes are not sufficient

Access Behaviour

- write amplification
- complex multi-threaded performance
- not all data required persistent

NVRAM in the *PAVE* project

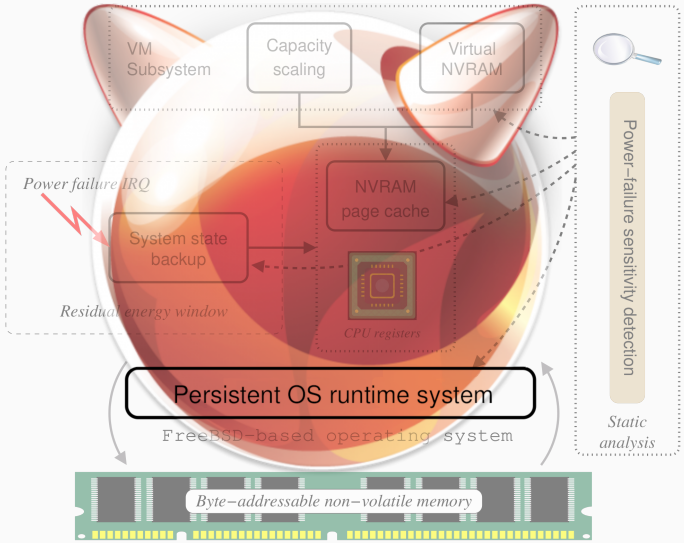
- integrate NVRAM into the VM-subsystem
  - utilize NVRAM capacity
  - balance between DRAM and NVRAM
- tolerate sudden blackouts
  - write back volatile state
  - create restauration point
  - resume computation on boot



NVRAM in the *PAVE* project

- integrate NVRAM into the VM-subsystem
  - utilize NVRAM capacity
  - balance between DRAM and NVRAM
- tolerate sudden blackouts
  - write back volatile state
  - create restauration point
  - resume computation on boot

→ recover state from NVRAM



Requirements for an allocator

Requirements for an allocator

- provide multiple page granularities
- minimal dependencies
- durability during power outage
- retrieval of data (by *key*)
- recover to consistent state
- reduce access to NVRAM

Requirements for an allocator

Requirements for an allocator

- provide multiple page granularities
- minimal dependencies
- durability during power outage
- retrieval of data (by *key*)
- recover to consistent state
- reduce access to NVRAM

Missing Properties in FreeBSD

- ✗ awareness of physical memory characteristics
- ✗ NVRAM capable locks
- ✗ minimal data structures
- ✗ consistent state after interruption

Requirements for an allocator

Requirements for an allocator

- provide multiple page granularities
- minimal dependencies
- durability during power outage
- retrieval of data (by *key*)
- recover to consistent state
- reduce access to NVRAM

Missing Properties in FreeBSD

- ✗ awareness of physical memory characteristics
- ✗ NVRAM capable locks
- ✗ minimal data structures
- ✗ consistent state after interruption

→ new transactional allocator required

Interface

```
int example(void) {
    int err, flags = NVA_FLAG_ALIGN | NVA_FLAG_ZERO;
    struct example {
        nv_ptr next;
        unsigned value;
    } *ptr;
    nv_ptr parent;

    nv_store_resolve("example", &parent);
    if (*parent == 0)
        err = nva_alloc(parent, 2, NVA_SIZE_4K, flags);
    ptr = *(struct example **)parent;

    if (ptr->next == 0)
        nva_alloc(&ptr->next, 1, NVA_SIZE_4K, flags);

    ptr->value++;
    struct memory_range_nv r = {ptr, ptr+1};
    nv_persist_range(&r);
    return ptr->value;
}
```

- recovery of previous allocation

Interface

```
int example(void) {
    int err, flags = NVA_FLAG_ALIGN | NVA_FLAG_ZERO;
    struct example {
        nv_ptr next;
        unsigned value;
    } *ptr;
    nv_ptr parent;

    nv_store_resolve("example", &parent);
    if (*parent == 0)
        err = nva_alloc(parent, 2, NVA_SIZE_4K, flags);
    ptr = *(struct example **)parent;

    if (ptr->next == 0)
        nva_alloc(&ptr->next, 1, NVA_SIZE_4K, flags);

    ptr->value++;
    struct memory_range_nv r = {ptr, ptr+1};
    nv_persist_range(&r);
    return ptr->value;
}
```

- recovery of previous allocation
- new allocation to volatile root obj

Interface

```
int example(void) {
    int err, flags = NVA_FLAG_ALIGN | NVA_FLAG_ZERO;
    struct example {
        nv_ptr next;
        unsigned value;
    } *ptr;
    nv_ptr parent;

    nv_store_resolve("example", &parent);
    if (*parent == 0)
        err = nva_alloc(parent, 2, NVA_SIZE_4K, flags);
    ptr = *(struct example **)parent;

    if (ptr->next == 0)
        nva_alloc(&ptr->next, 1, NVA_SIZE_4K, flags);

    ptr->value++;
    struct memory_range_nv r = {ptr, ptr+1};
    nv_persist_range(&r);
    return ptr->value;
}
```

- recovery of previous allocation
- new allocation to volatile root obj
- new allocation to persistent obj

Interface

```
int example(void) {
    int err, flags = NVA_FLAG_ALIGN | NVA_FLAG_ZERO;
    struct example {
        nv_ptr next;
        unsigned value;
    } *ptr;
    nv_ptr parent;

    nv_store_resolve("example", &parent);
    if (*parent == 0)
        err = nva_alloc(parent, 2, NVA_SIZE_4K, flags);
    ptr = *(struct example **)parent;

    if (ptr->next == 0)
        nva_alloc(&ptr->next, 1, NVA_SIZE_4K, flags);

    ptr->value++;
    struct memory_range_nv r = {ptr, ptr+1};
    nv_persist_range(&r);
    return ptr->value;
}
```

- recovery of previous allocation
- new allocation to volatile root obj
- new allocation to persistent obj
- enforce write to NVDIMM

Transactions

```
// ar: Allocated Range
// container: nv_ptr
ar->range.end = free_range.end;
ar->range.start = free_range.start;
nv_persist_range(ar);
```

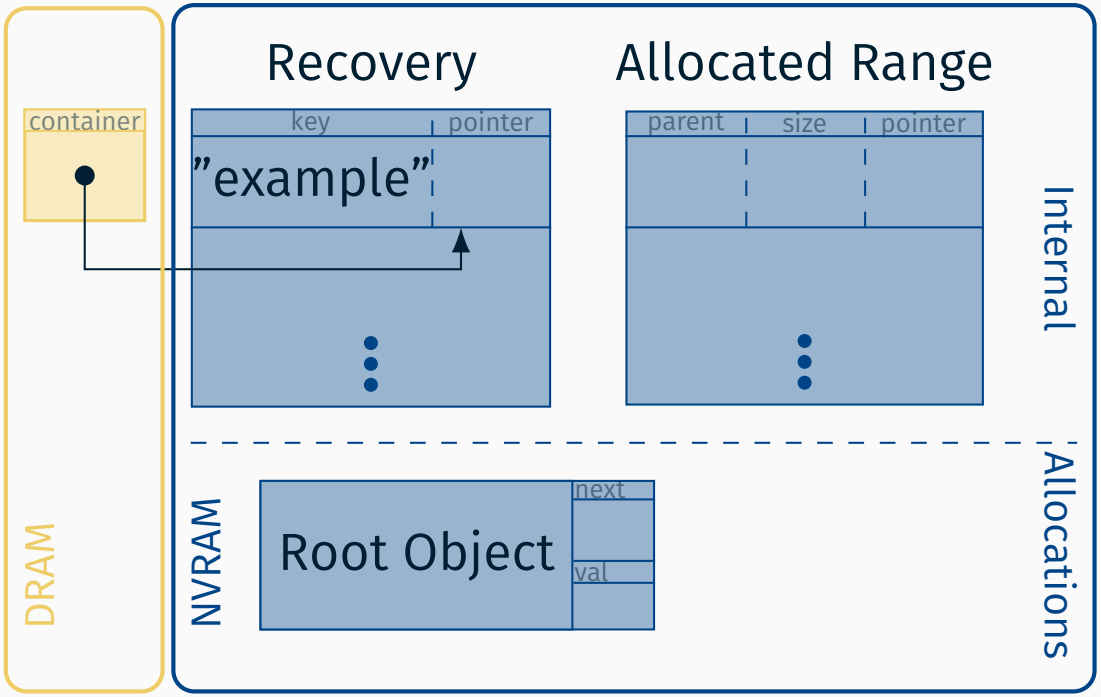
1

```
ar->parent = container;
nv_persist_range(ar);
```

2

```
*container = range->start;
nv_persist_range(container);
```

3



Transactions

```
// ar: Allocated Range
// container: nv_ptr
ar->range.end = free_range.end;
ar->range.start = free_range.start;
nv_persist_range(ar);
```

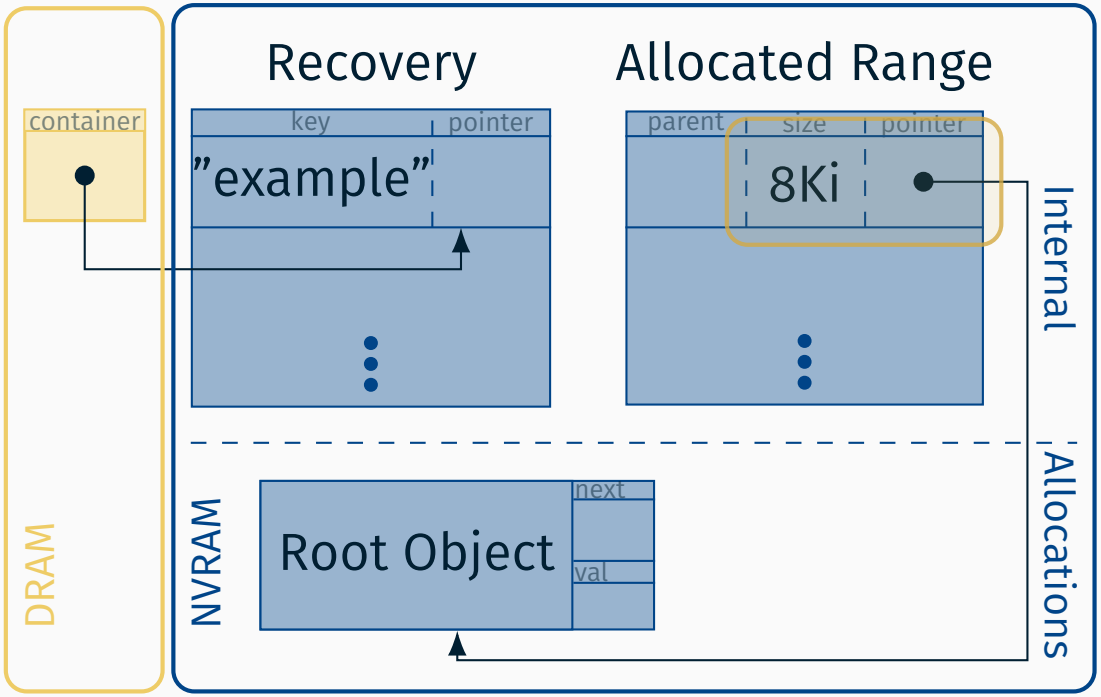
1

```
ar->parent = container;
nv_persist_range(ar);
```

2

```
*container = range->start;
nv_persist_range(container);
```

3



Transactions

```
// ar: Allocated Range
// container: nv_ptr
ar->range.end = free_range.end;
ar->range.start = free_range.start;
nv_persist_range(ar);
```

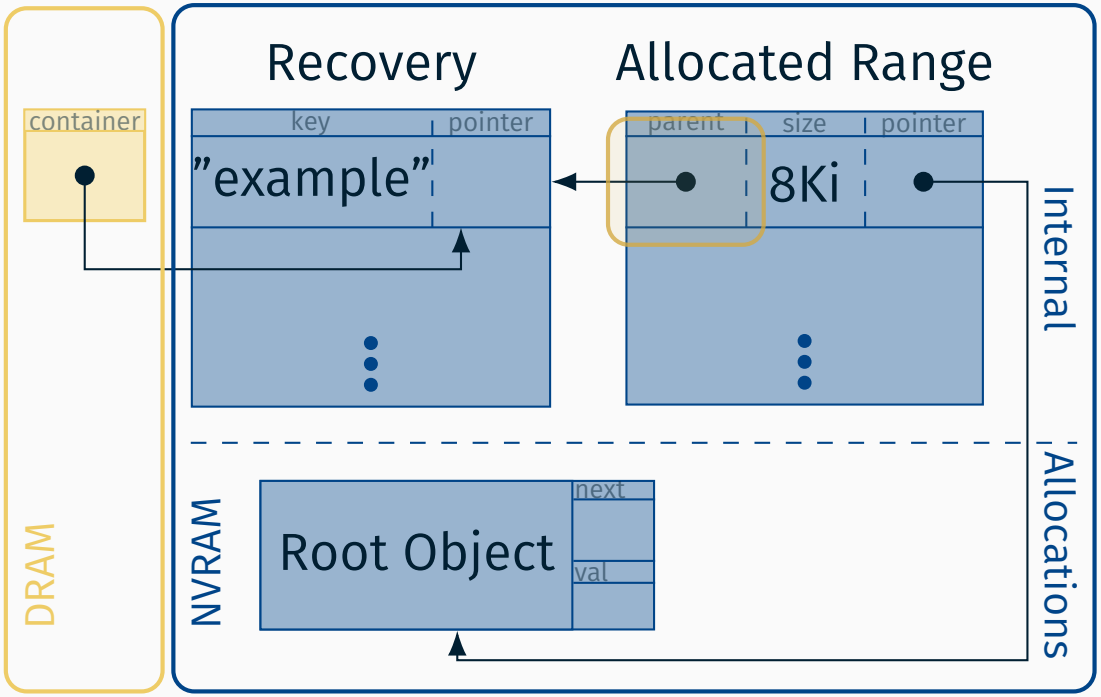
1

```
ar->parent = container;
nv_persist_range(ar);
```

2

```
*container = range->start;
nv_persist_range(container);
```

3



Transactions

```
// ar: Allocated Range
// container: nv_ptr
ar->range.end = free_range.end;
ar->range.start = free_range.start;
nv_persist_range(ar);
```

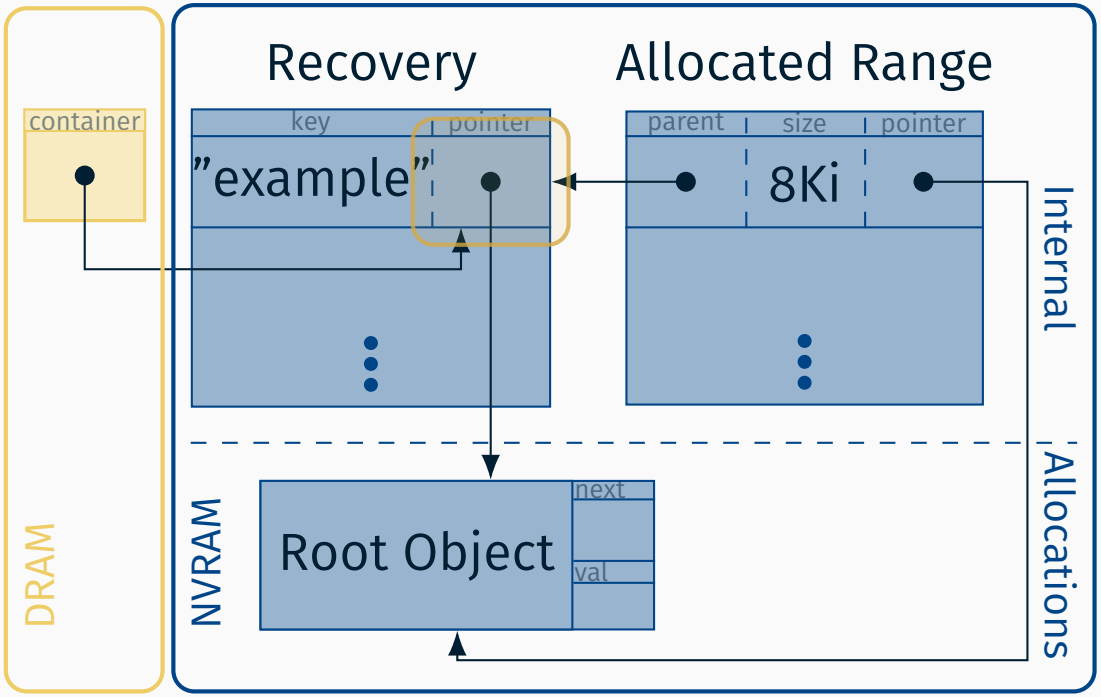
1

```
ar->parent = container;
nv_persist_range(ar);
```

2

```
*container = range->start;
nv_persist_range(container);
```

3



Transactions

```
// ar: Allocated Range
// container: nv_ptr
ar->range.end = free_range.end;
ar->range.start = free_range.start;
nv_persist_range(ar);
```

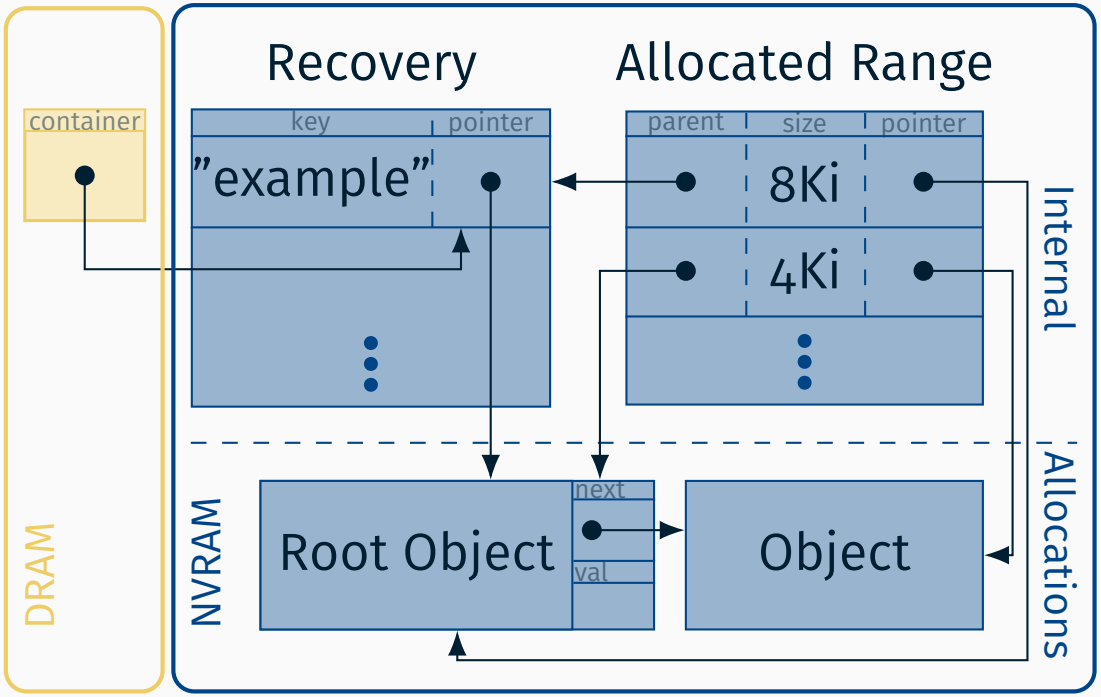
1

```
ar->parent = container;
nv_persist_range(ar);
```

2

```
*container = range->start;
nv_persist_range(container);
```

3



Conclusion

With *NVall* we have

- ✓ very few dependencies
- ✓ support for varying granularity
- ✓ durability in case of crash
- ✓ retrieval of previous state

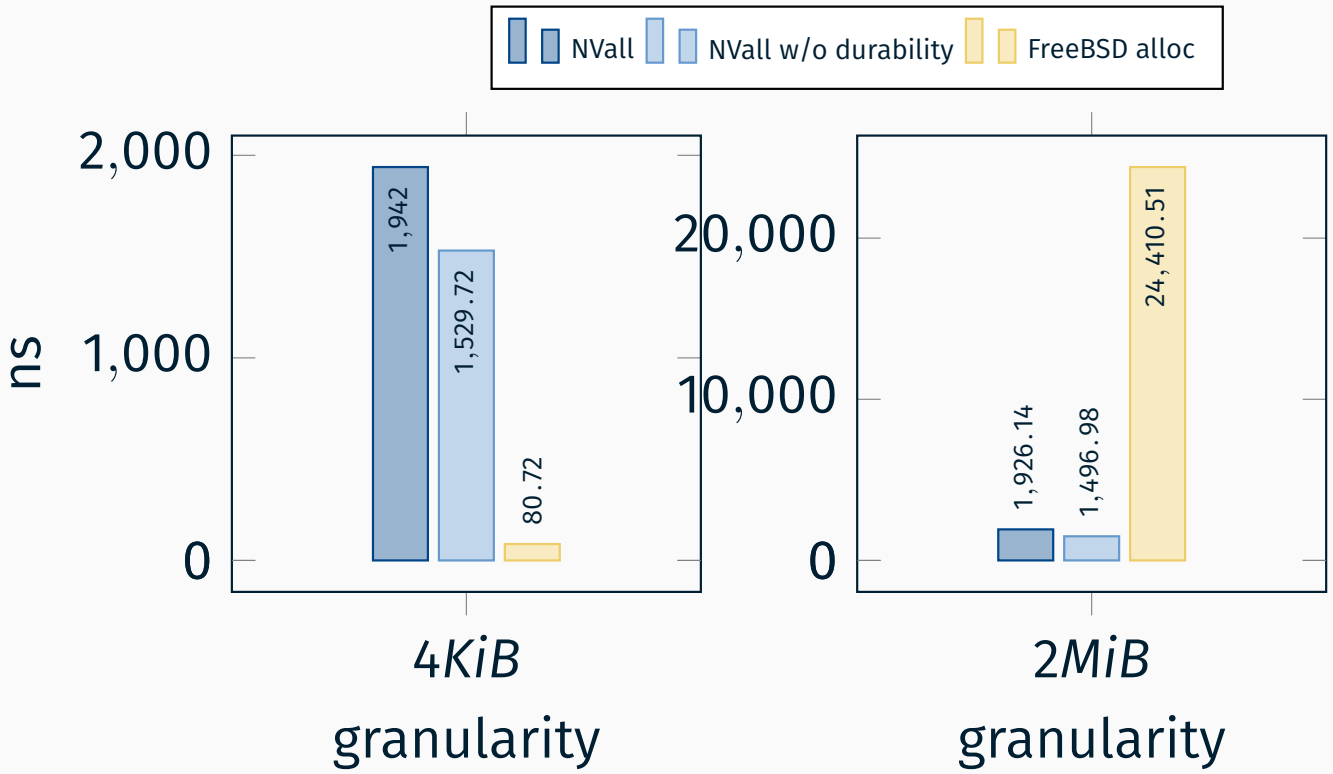
We can build

- → suspend/resume with NVRAM
- → statistic logging of early boot/shutdown

Source code available at:
https://doi.org/10.5281/zenodo.8364439

Appendix

Eval


NVall
NVall w/o durability
FreeBSD alloc
ns
2,000
1,000
0
1,942
1,529.72
80.72
4KiB
granularity
20,000
10,000
0
1,926.14
1,496.98
24,410.51
2MiB
granularity