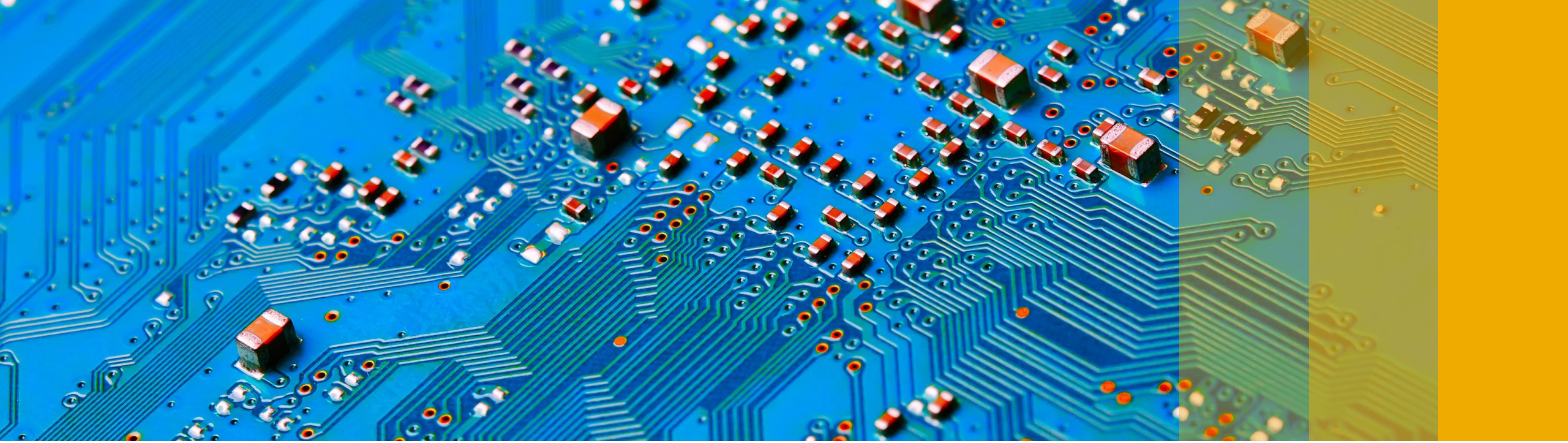


Memory Management Techniques for Large-Scale Persistent-Main-Memory Systems [VLDB 2017]

Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, Grégoire Gomes

Non-Volatile Memories Workshop – March 12, 2018

SAP Run Simple

Dresden Database Systems Group

intel

Motivation

- NVM can replace both main memory and storage
  → single-level database storage architecture without I/O
- Fail-safe persistent NVM memory management is *conditio sine qua non* for enabling this novel architecture paradigm
- Existing persistent allocators are general-purpose and do not address the versatile needs of database systems
- We present PAllocator, a highly scalable fail-safe persistent allocator

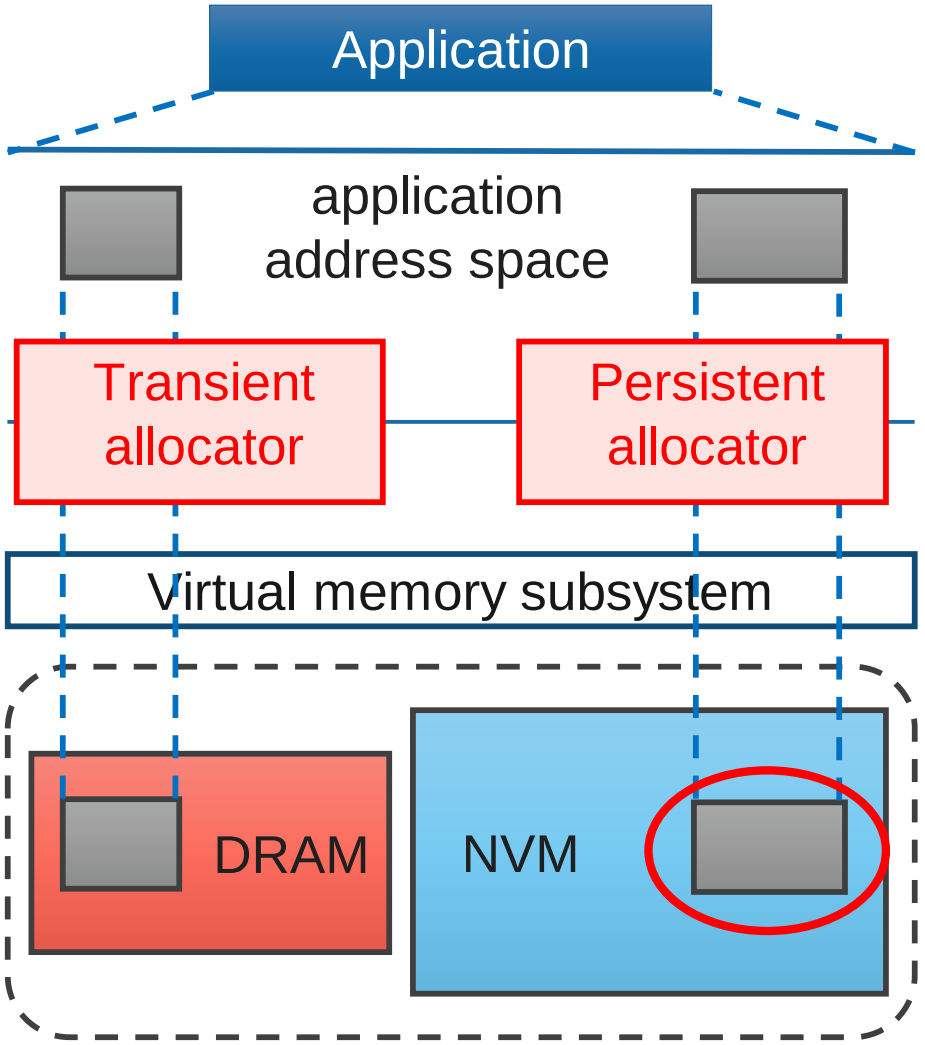
Outlook

- What is a persistent allocator?
- PAllocator’s design decisions
- Experimental evaluation
- Conclusion

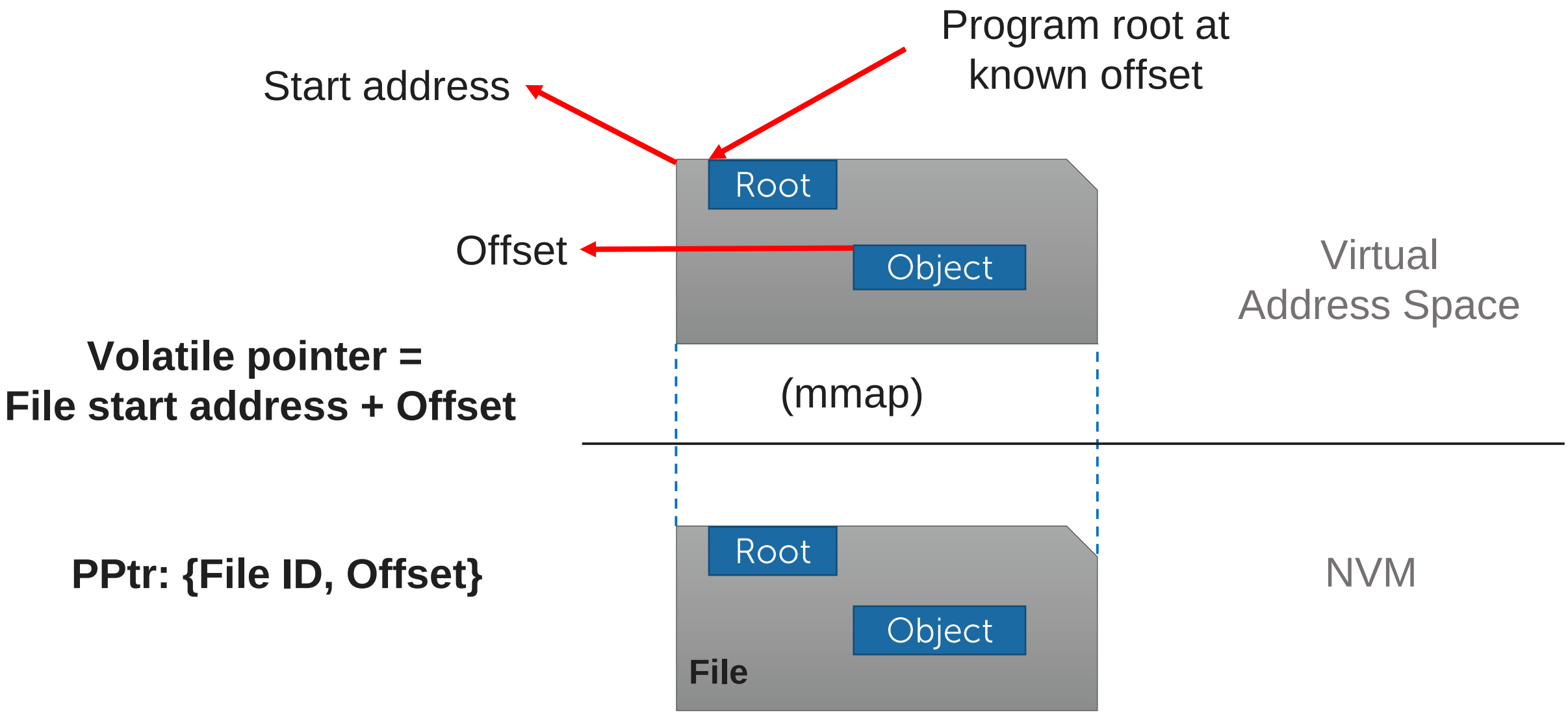
What Characterizes a Persistent Allocator?

A persistent allocator must:

1. Provide a recoverable addressing scheme
2. Avoid persistent memory leaks



1. Recoverable Addressing Scheme



2. Preventing Memory Leaks

```
pptr = allocate(size);
persist(&pptr);
```

→ Traditional interface has a “blind spot”

Reference passing

→ `allocate(PPtr &pptr, size_t allocSize)`

`pptr` is owned by the data structure

PAllocator Design

We explore the following design dimensions

1. Pool structure (single file vs. multiple files)
2. Allocation strategies
3. Concurrency Handling
4. Persistent fragmentation

We do not consider garbage collection

We assume hardware-managed wear-leveling

1. Pool Structure: Single Vs. Multiple Files

Pool as Single File

Pros

- 8-byte persistent pointers possible
- Easier to implement

Cons

- Hard to shrink
- Huge block allocation a problem

Pool as Multiple Files

Pros

- Easier to grow and shrink
- Easy, fragmentation-free huge allocation handling

Cons

- 16-byte persistent pointers

Multiple files better suited for database systems

2. Allocation Strategies

Three allocation strategies

→ One file per allocation

→ Segregated-fit for small blocks (e.g., `< 4 KB`)

→ Best-fit for medium and large blocks (e.g., `[4 KB, 16 MB)`)

One file per allocation not realistic…

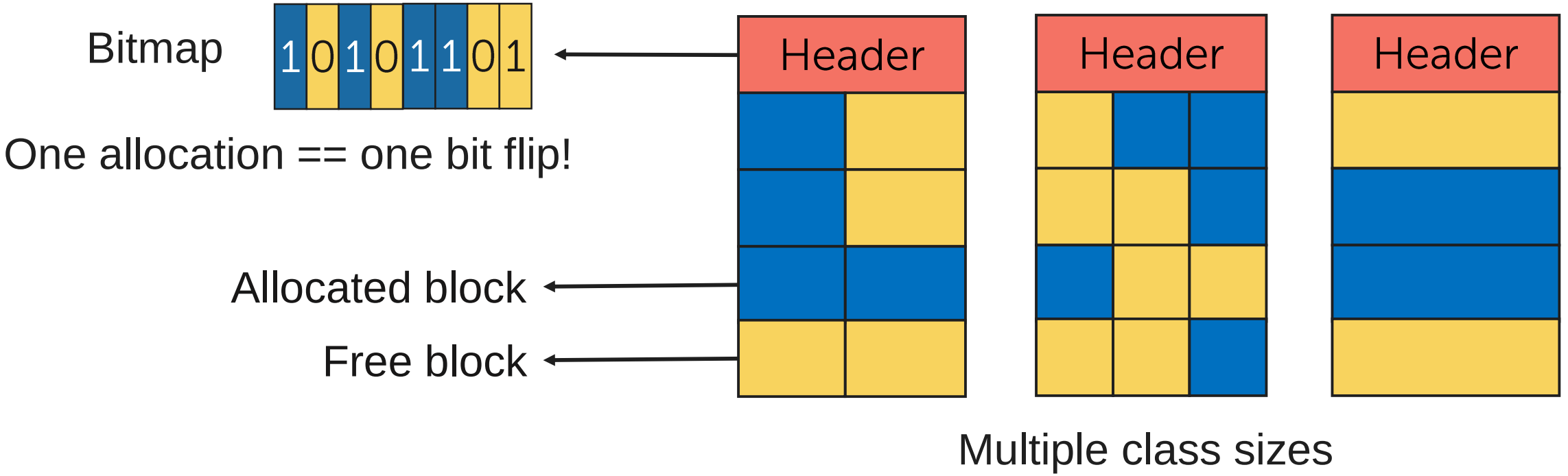
- Significant overhead and wasted memory for small blocks
- Filesystem might struggle to handle huge number of files

except for huge blocks!

→ Fragmentation handling pushed to filesystem

Segregated-Fit Allocation Strategy

Fixed-size memory chunk, e.g., 8 KB, divided into fixed-size blocks



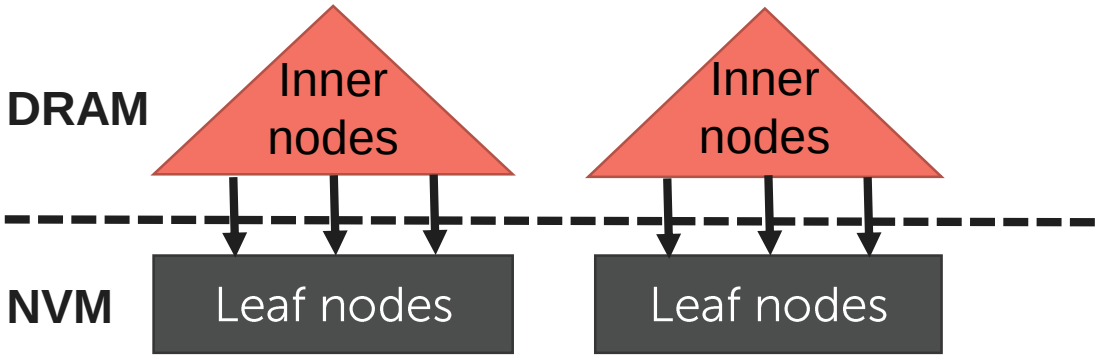
→ **Reduced fragmentation with moderate number of class sizes**
→ **Not suitable for larger block allocations**

2. Allocation Strategies: Best-Fit Allocation Strategy

Allocate multiple of a predetermined size (e.g., system page size)

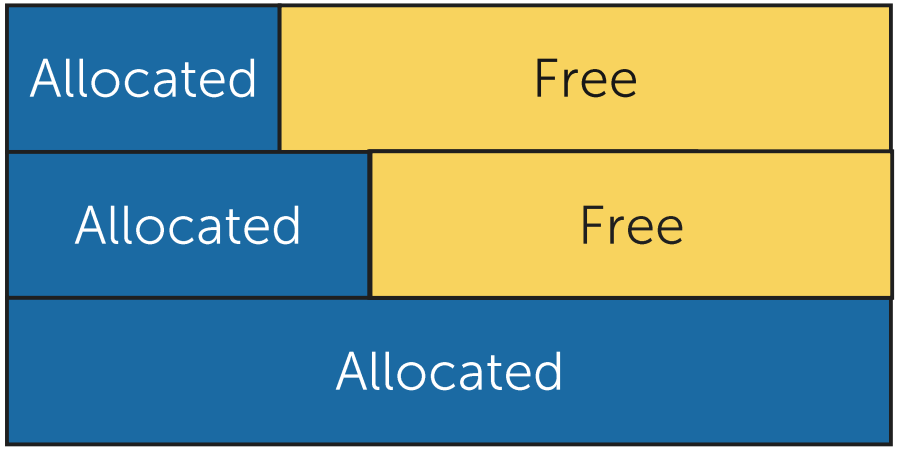
Allocation

Free blocks index sorted by block size



Coalescing

Global block index sorted by block offset



Segment (e.g., 128 MB)

Indexes implemented with the FPTree, a hybrid NVM-DRAM B+-Tree [SIGMOD'16]

→ **Suitable for large blocks** → **Prone to fragmentation**

3. Concurrency Handling

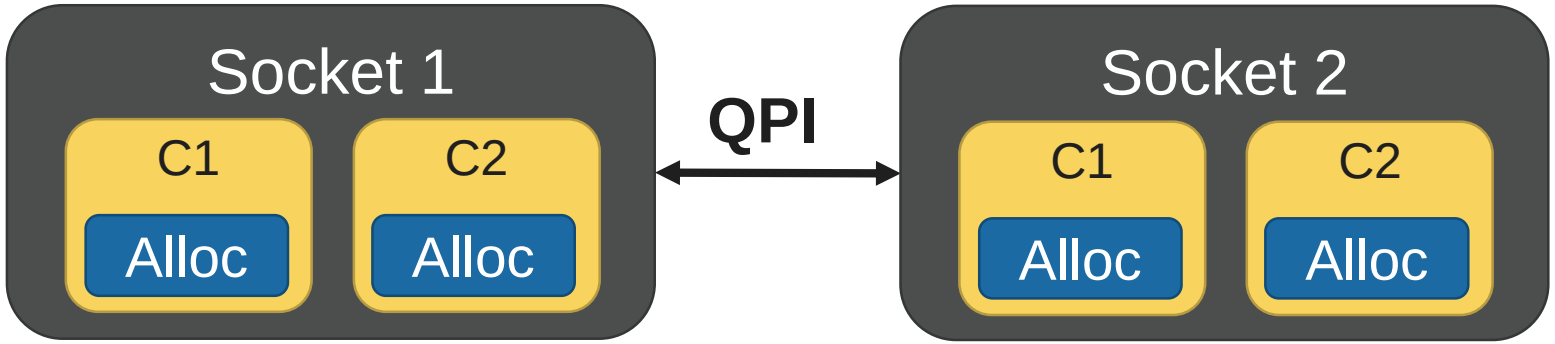
Thread-local allocation → One allocator object per thread

- The standard in general-purpose allocators
- Used for small block allocations
  → Local allocator requests chunks from global pool
- Need to be merged with global pool when thread terminates
- Does not scale under high concurrency
  → Frequent chunk requests to the global pool

3. Concurrency Handling

Core-local allocation → One allocator object per physical core

- Local allocators request large files from global pool



- Robust performance under high concurrency
  → Stable local allocators → Greedy

4. Persistent Fragmentation

Restart is a last resort, but valid way of defragmenting volatile memory
→ does not apply to NVM

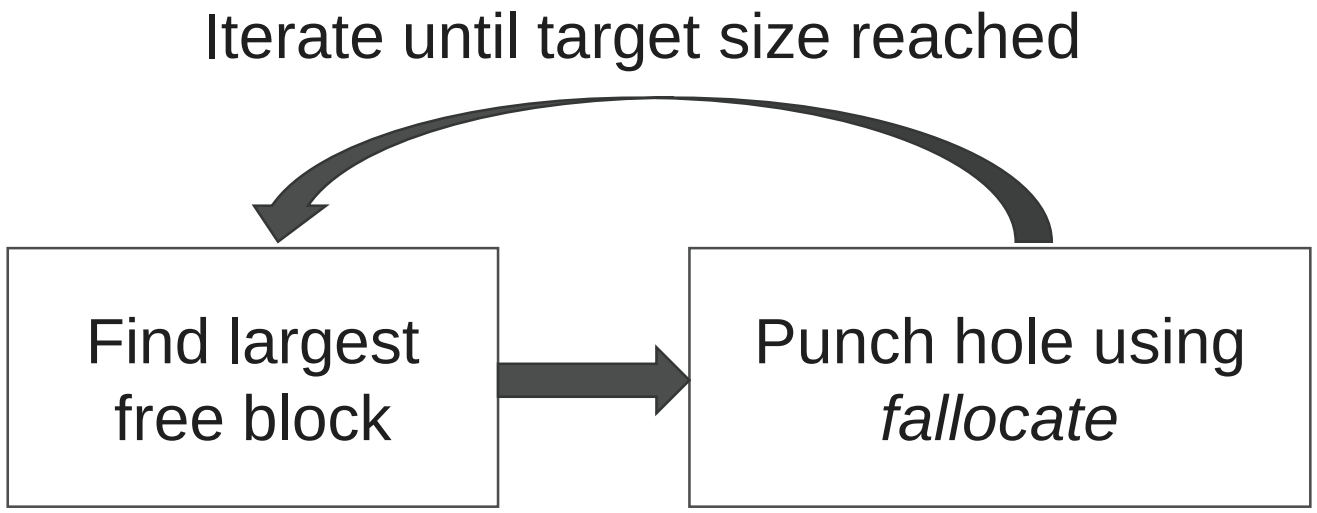
File system solutions do not apply to NVM
- File systems benefit from an additional indirection layer
- NVM is directly accessed with load/store instructions

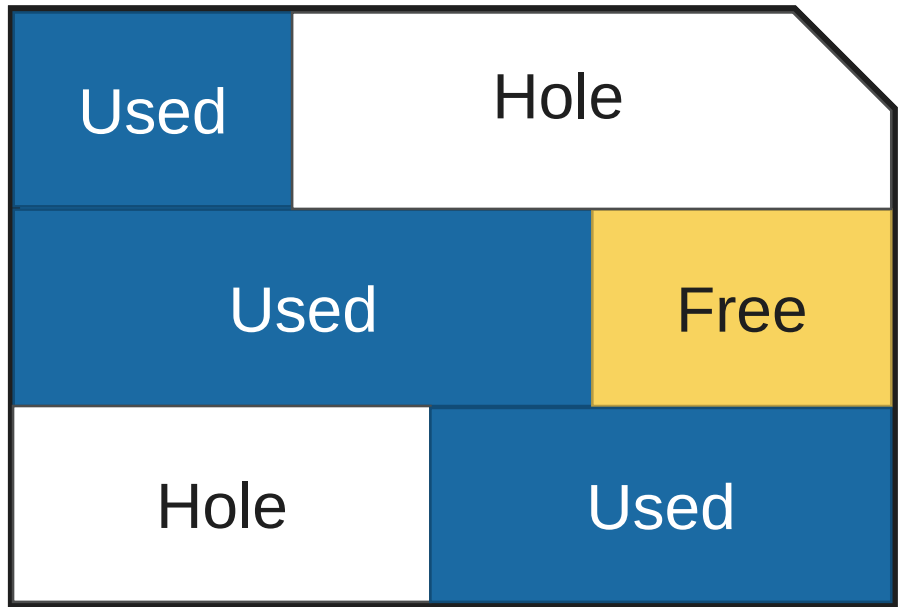
Need new defragmentation mechanisms

4. Persistent Fragmentation

Most file systems have support for sparse files

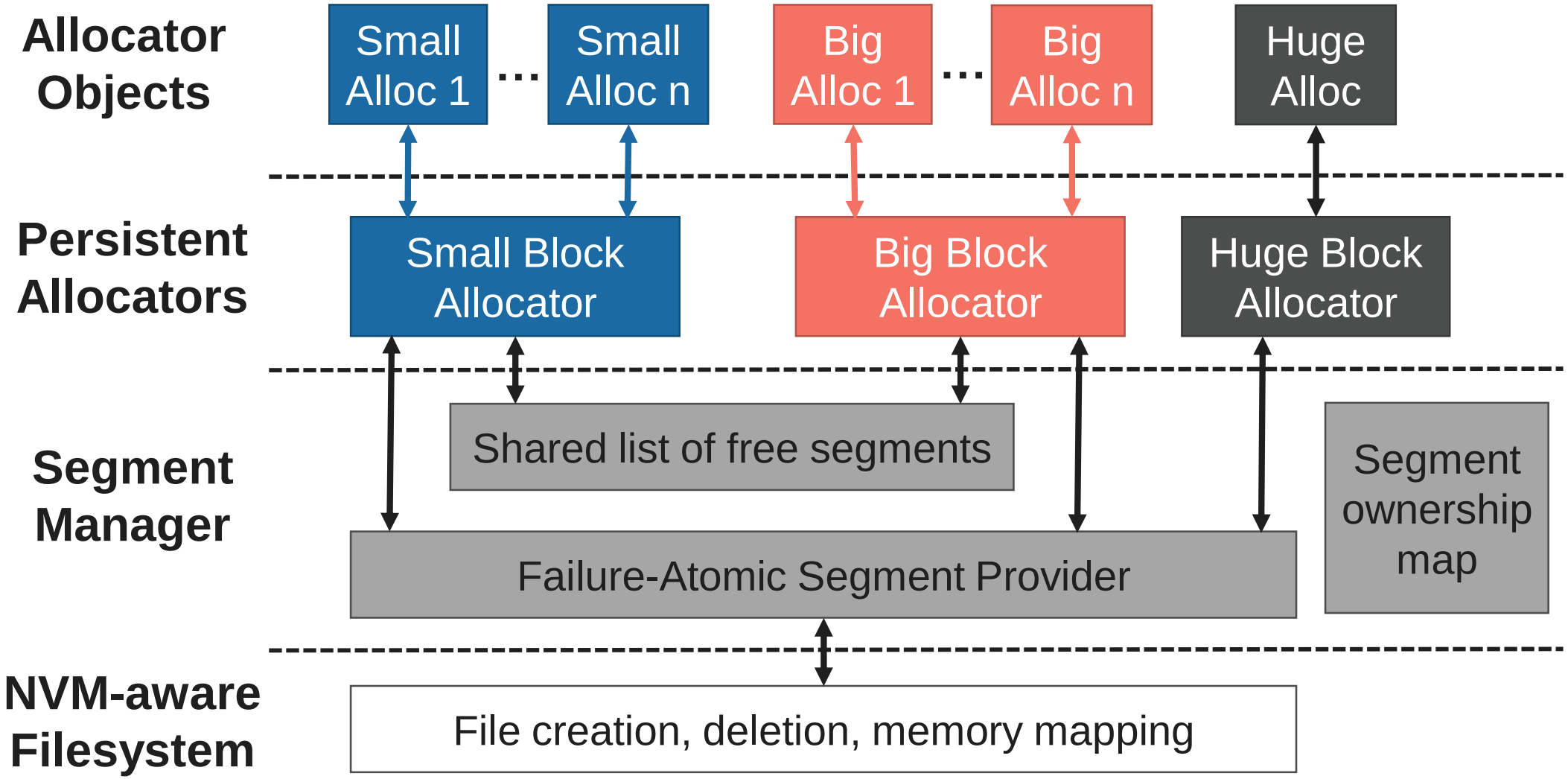
Defragmentation idea: **Punch holes** in free blocks



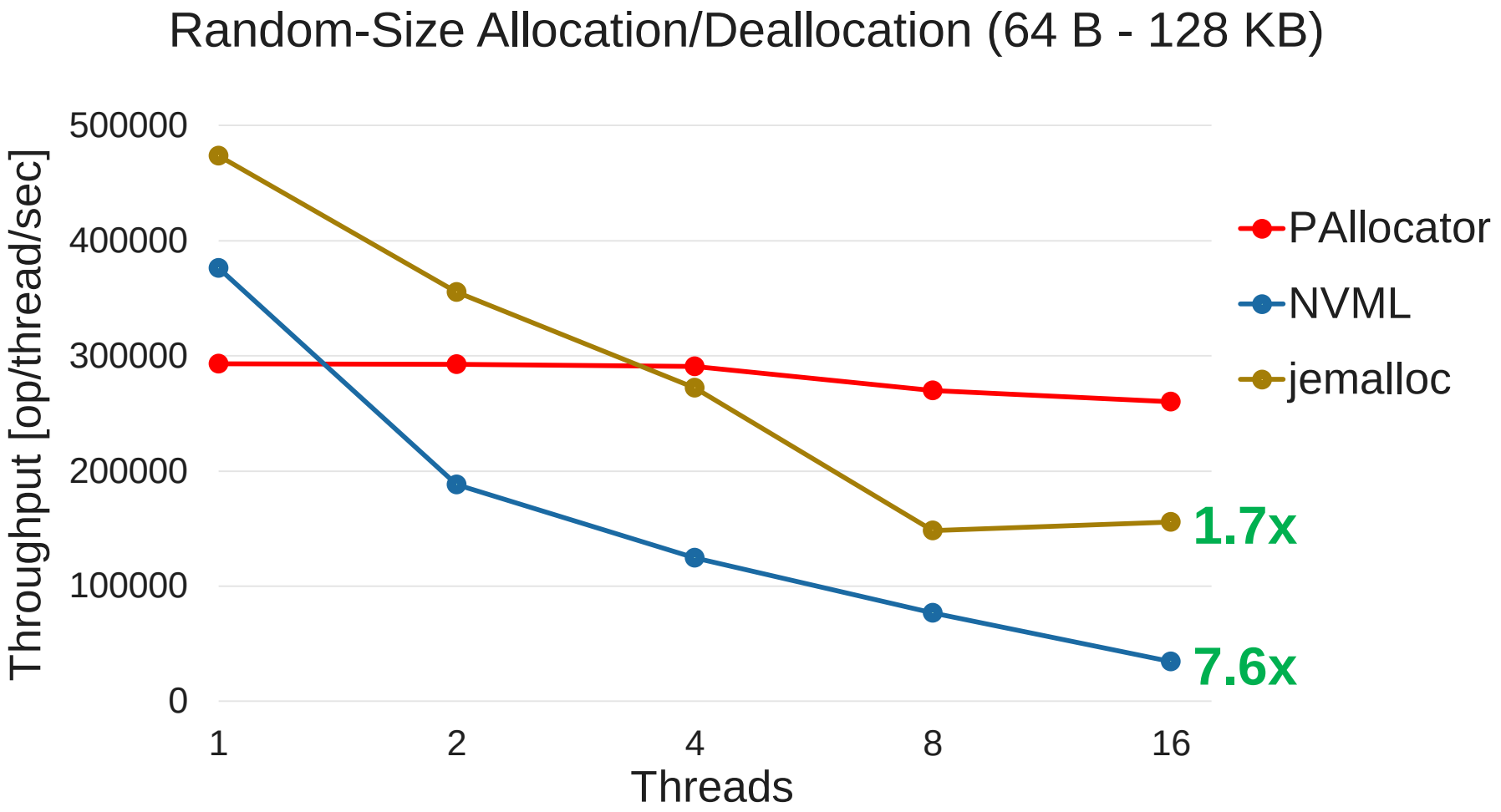


Must keep file size unchanged to maintain validity of offsets

PAllocator: Architecture Overview

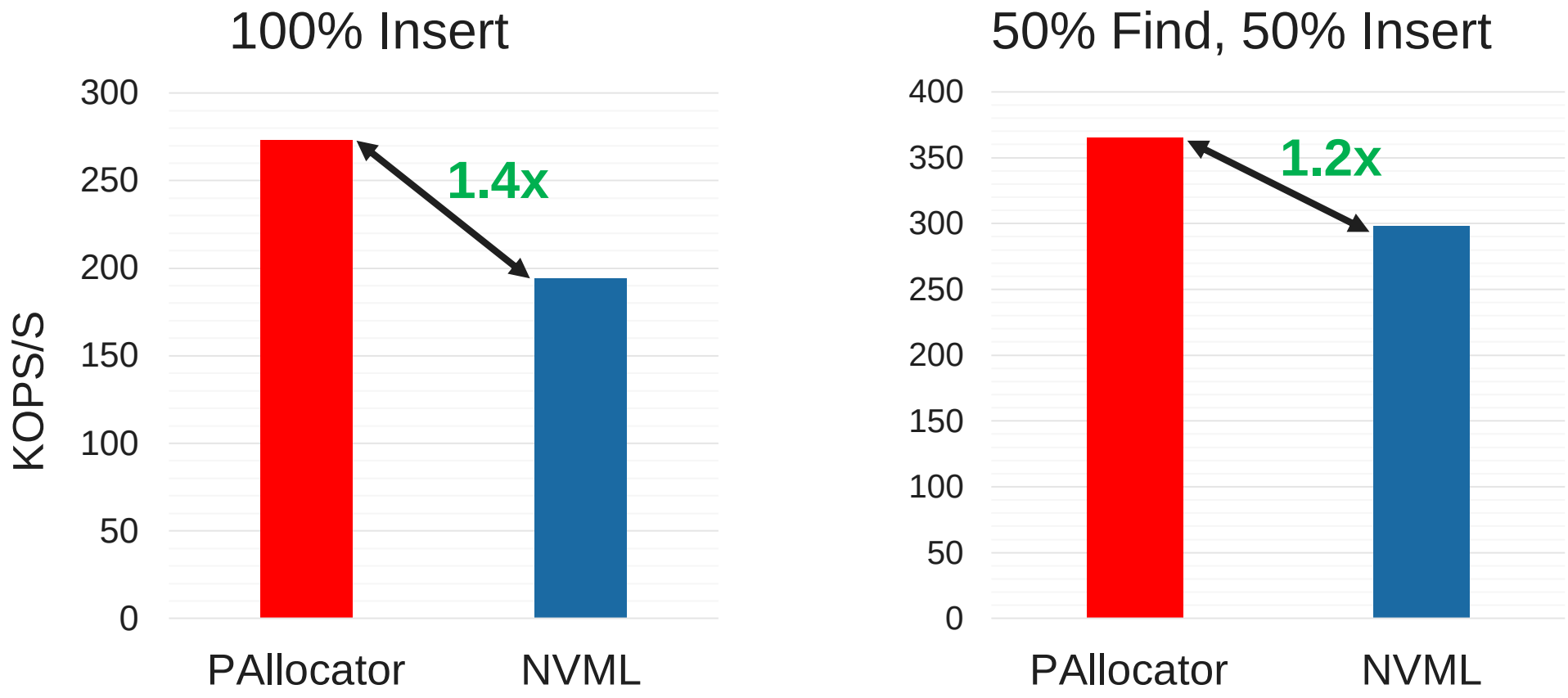


PAllocator Performance Evaluation



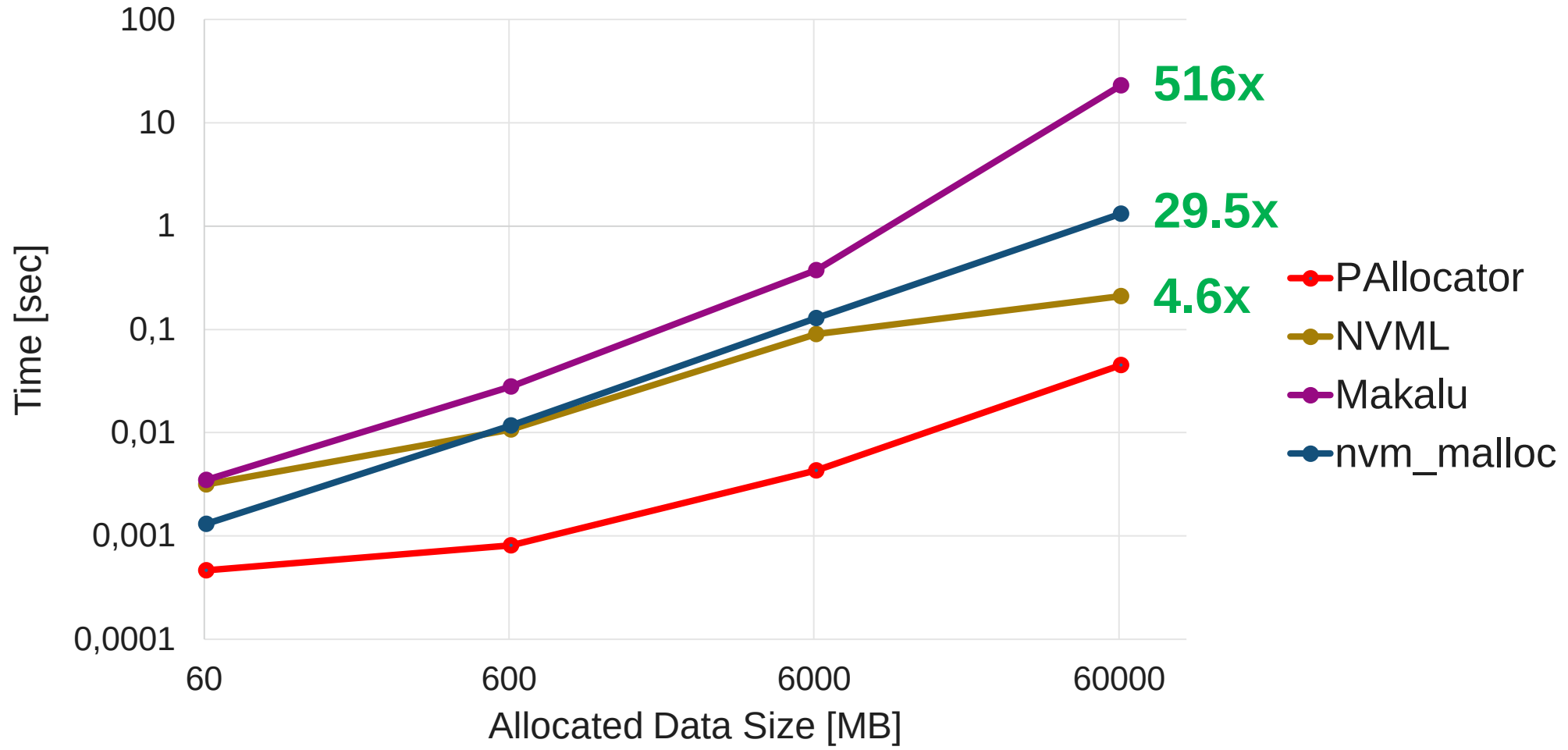
PAllocator scales nearly linearly

Allocator Performance Impact on the FPTree



Persistent allocators do impact database performance

Allocator Recovery Time



1 TB → PAllocator (0.75s), NVML (3.5s), Makalu (394.5s), nvm_malloc (22.5s)

Conclusion

NVM has the potential to disrupt database storage architecture

➢ Memory management is a necessary building block

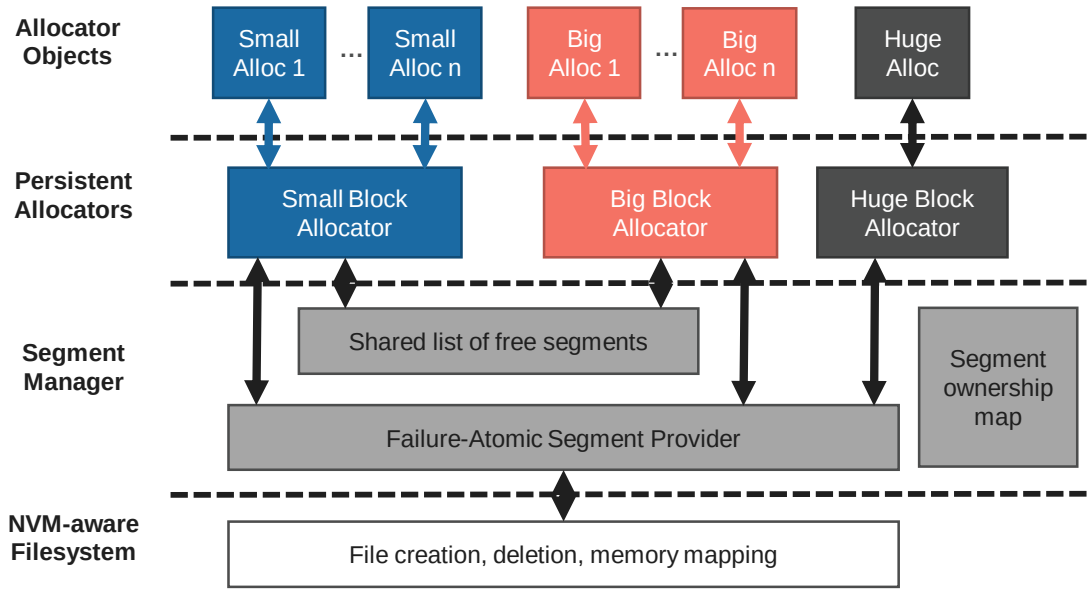
We presented **PAllocator**:

→ Designed for large NVM systems

→ Highly scalable

→ Fast recovery

→ Defragmentation capability



State-of-the-Art

Allocator	Purpose	Pool structure	Allocation strategies	Concurrency handling	Garbage collection	Defragm-entation	Source
Mnemosyne	General	Multiple files	Segregated-fit + best-fit	Thread-local for small blocks	Yes	No	ASPLOS‘11
NV-Heaps	General	Single file	Undefined	Thread-local	Yes	No	ASPLOS‘11
nvm_malloc	General	Single file	Segregated-fit + best-fit	Thread-local for small blocks	No	No	ADMS‘15
NVML	General	Single file	Segregated-fit + best-fit	Thread-local for small blocks	No	No	http://pmem.io/nvml/
Makalu	General	Single file	Segregated-fit + best-fit	Thread-local for small blocks	Yes (offline)	No	OOPSLA‘16
PAllocator	**Large systems**	**Multiple files**	**Segregated-fit + best-fit + file**	**Core-local**	No	**Yes**	VLDB‘17

For completeness: NVMalloc and Walloc focus on wear-leveling

Salient differences in design decisions