Twizzler: a Data-Centric OS for Non-Volatile Memory

Before we start, some Q&A.

Q. Non-Volatile Memory (Persistent memory) is good, but why should anyone care about NVM/PM?

A. i. PM is faster.

Byte-addressable Nice!

Some experiment in Redis shows that **PMDK-Redis > PMEM Block I/O > SSD** in Throughput

ii. PM is persistent.

AOF(append-only-file) in Redis guarantee the safety of data. **But what is the cost?**

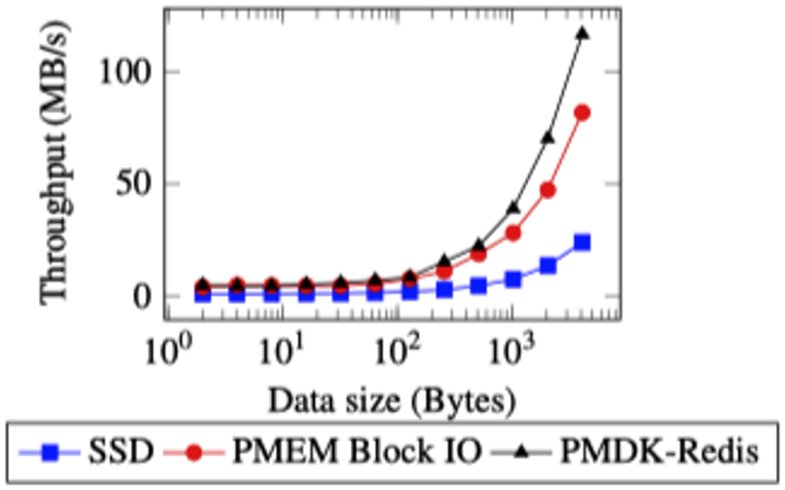


Figure 1: Redis throughput comparison against memtier

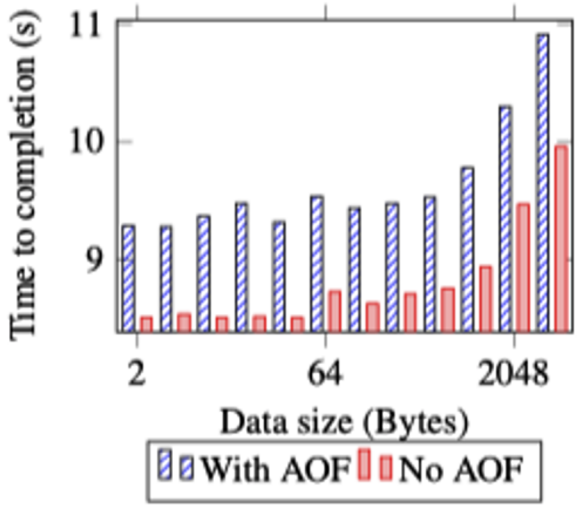
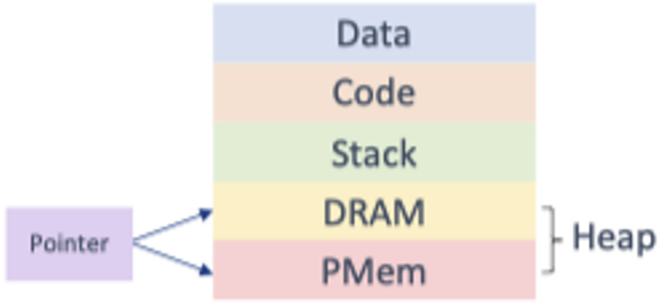
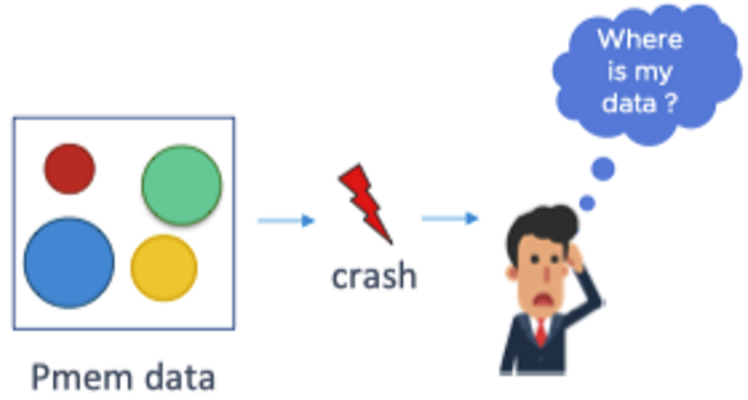


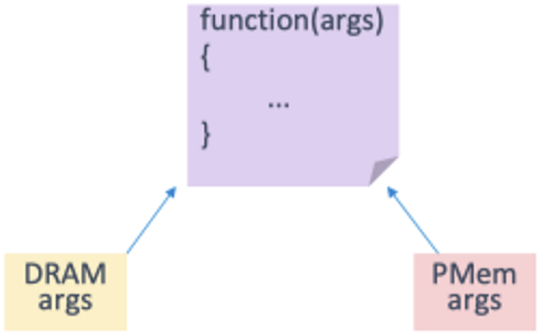
Figure 2: Redis-server runtime on nullfsvfs

Design Goals

1. Single type system - keep friendly to prograr
2. Two heaps
3. Use native pointers
4. Access PMem data across restarts
5. Crash-consistent data updates
6. Reuse functions
7. PMem updates outside transactions







Runtime Design

Pointer Swizzling

- Why use this mechanism?
  - Linux exposes PM to application as a file, so PM may be mmap’ d to different address.
  - But PM is persistent, remember that?
- What does Pointer Swizzling do?
  - Common scenario, mmap PM to the same addre
  - Update pointer transactionally.

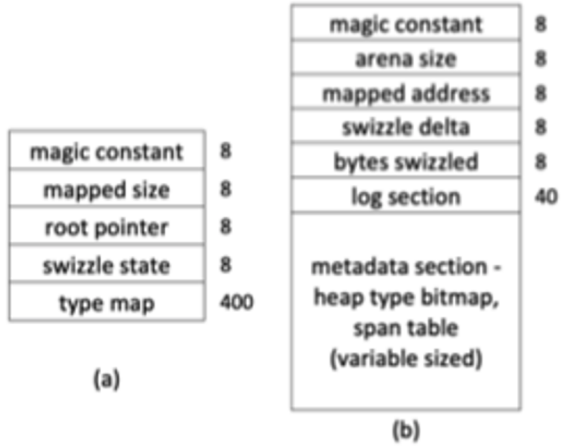
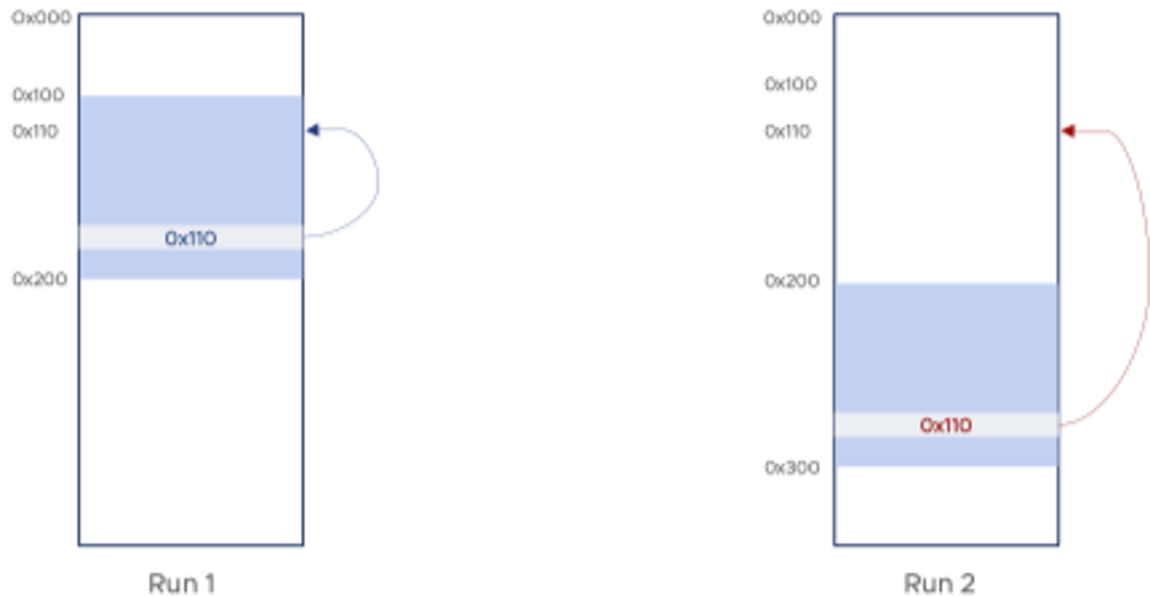


Figure 3: (a) pmem file header (b) arena header layout. Storage space in bytes adjacent to each field.

The novelty is the minimal amount of additional metadata that they log.

Pointer Swizzling

- PMem file map address change makes all stored pointers invalid!



Transactions

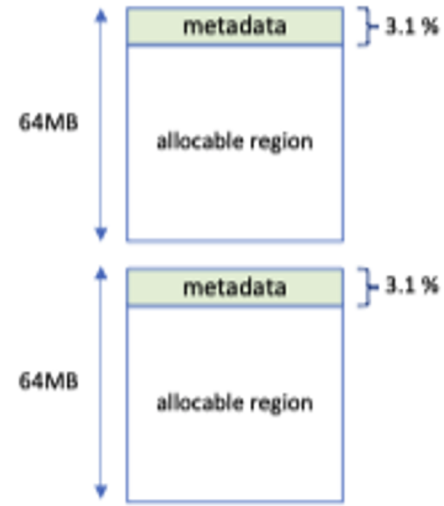
- Crash Consistent updates
- Atomicity – All or nothing visibility
- Consistency – Application defined
- Isolation - Use Go mutex
- Durability – Auto persisted on success

```
// add new node to tail
func addNode (tail  *node) *node {
  n := pnew(node)                  // <-
  txn("undo") {                    // <-
    mutex.Lock()
    n.prev = tail
    updateTail(tail, n)
    mutex.Unlock()
  }                                // <-
  return n
}

func updateTail(tail, n *node) {
  txn("undo") {                    // <-
    tail.next = n
  }                                // <-
}
```

Heap Recovery

1. Restore memory allocator volatile state
2. Swizzle pointers (if needed)
3. Run garbage collector



Some points

- Should not create new persistent data type Interesting
  - which may complicate the programming model.
  - (Maybe the same reason that they use Go rather than C/C++)
- Function reuse Disappointing
  - Just available for data in both volatile and persistent memory.
  - As for function call and handle,
  - Instead of cloning the function for transactional access
  - Maintain a per Go-routine handle and ask the user to wrap any potentially transactional code within a txn code block.

Persistent data should be operated on directly and like memory

- Traditional I/O interface for persistent data
  - heavy kernel involvement
  - a need for data serialization
  - complexity in data sharing requiring the overhead of pipes or management cost of shared virtual memory
- Target :
  - Remove the kernel from the persistence path
  - Design for pointers that last forever

Data-Centric Approach

- Use **persistent pointer** to name a word of persistent memory
- Kernel removed.
    - nothing fundamentally connects a virtual address space and a security context
    - how threads access data?
    - what data they access?
- Target these Constraints with Twizzler
    - a low level persistent object model
    - a persistent pointer design
    - an address space mechanism called views
    - a security context mechanism

Existing approaches?

Center for Research in Storage Systems

POSIX

Explicit persistence and data access

Multiple forms of data

Kernel involvement

mmap helps, but does not solve the virtual memory problem

PMDK

No OS support

Data sharing is hard

Slow pointers

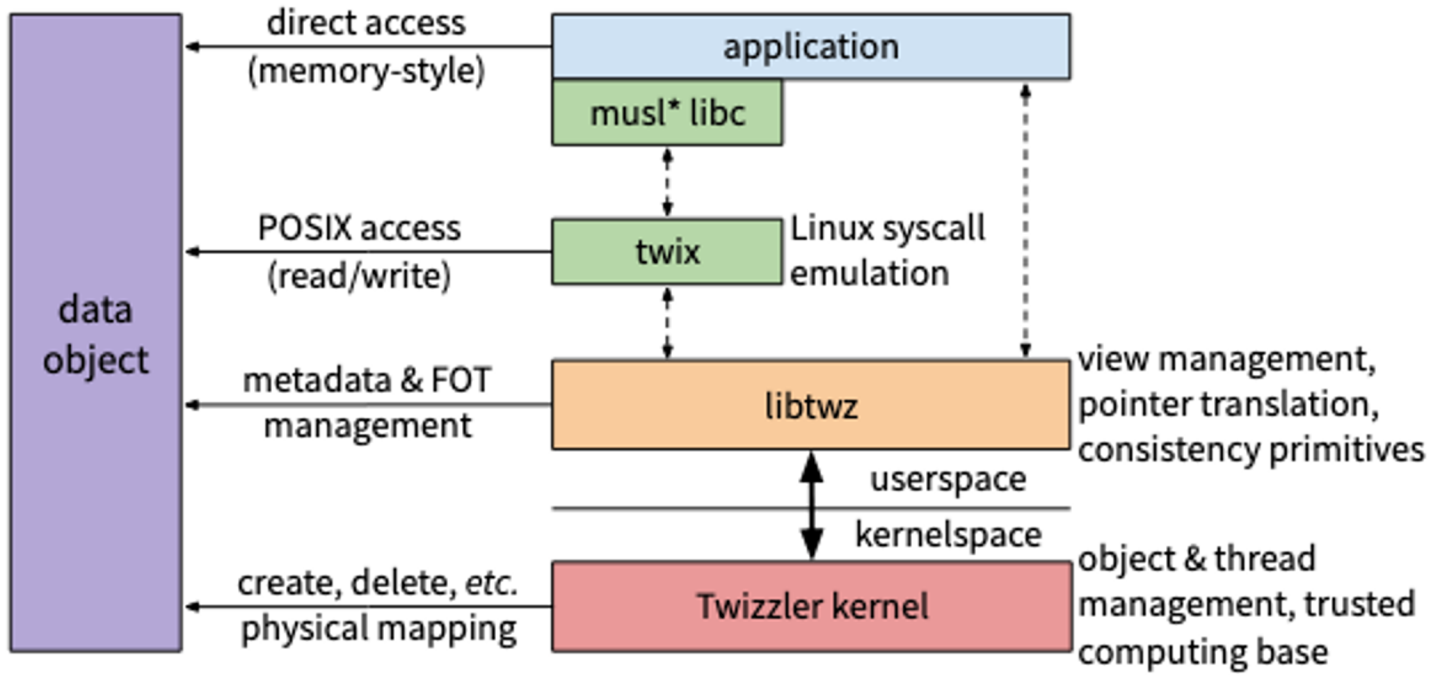
Twizzler's goals

Little kernel involvement

Pervasive support (security, sharing)

Low overhead persistent pointers

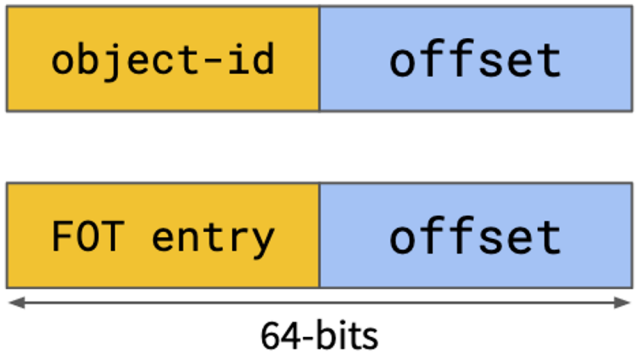
Design Overview

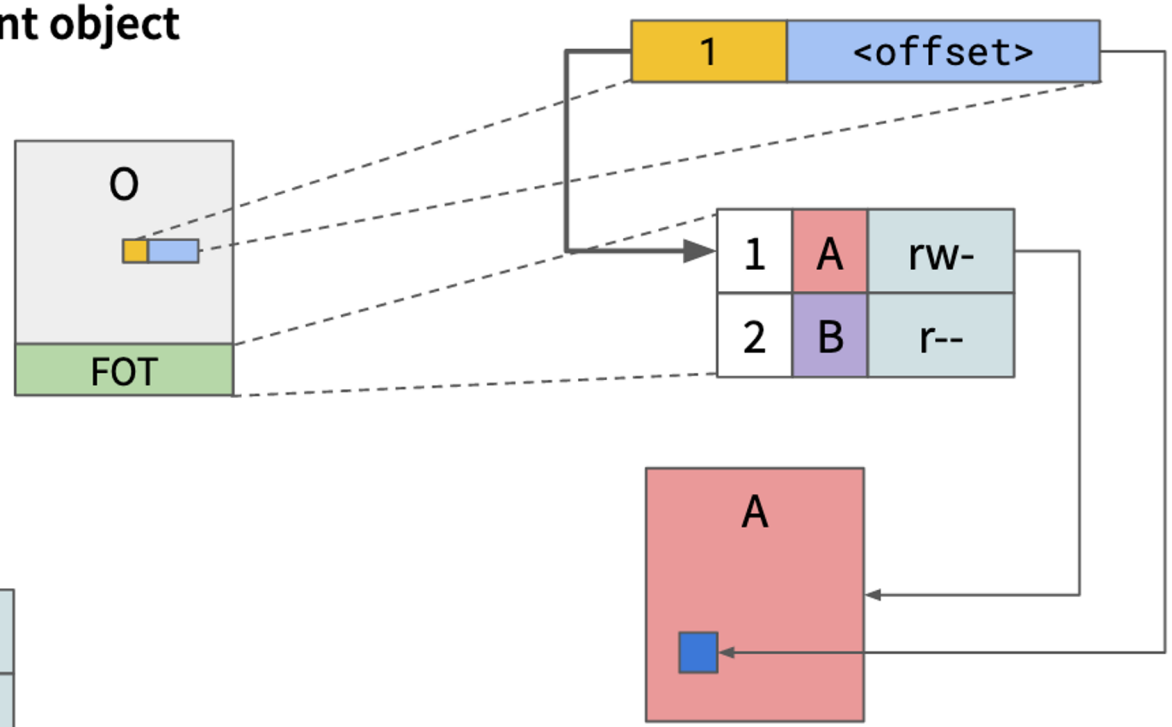


* modified musl to change linux syscalls into function calls

Persistent Pointers

Pointers may be *cross-object*: referring to data within a different object





Foreign Object Table

1	object ID or Name	Name Resolver	flags
2	object ID or Name	Name Resolver	flags
	...		

Object Layout

FOT entry of >0 means "cross-object"—points to a different object.