Metall: An Allocator for Persistent Memory

Keita Iwabuchi, Roger Pearce, Maya Gokhale

LLNL-PRES-817002
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

Background
Large-scale Data Analytics

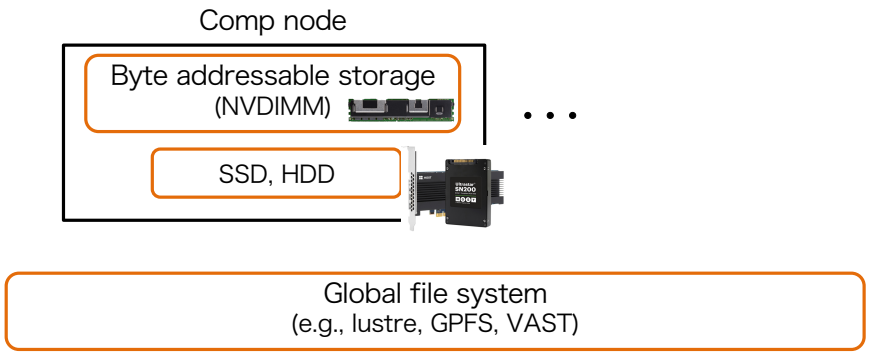
- High volume data analytics is one of key challenges in exascale
- Data ingestion
  - Indexing and partitioning data with analytics-specific data structures
    - e.g., read raw graph data from text files, and transform into a graph data structure
  - Often more expensive than analytics
  - The same data (or derived data) is re-ingested frequently
    - e.g., run multiple analytics to the same data, test different parameters, develop/debug analytics program



Background

Persistent Memory (PM) in HPC

- Substantial performance improvements and cost reductions
- Many HPC systems have PM devices to leverage them in large data processing with reduced cost and power consumption



Supercomputers w/ PM

- Sierra
- Summit
- Aurora
- Mammoth
- Fugaku (RIKEN, Japan)

.
.
.

Once constructed, data structures can be re-analyzed and updated beyond a single process lifecycle

Background
Data Serialization

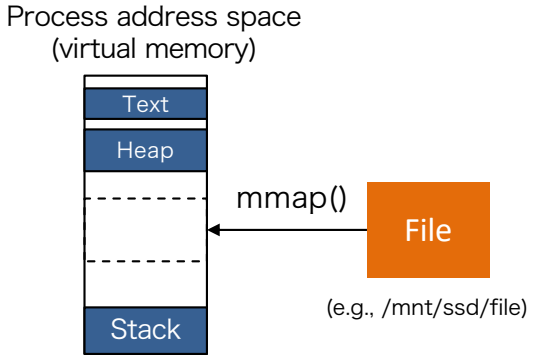
- Data serialization is expensive
  - Dismantling and assembling large complex data structures are expensive in terms of performance and programming cost

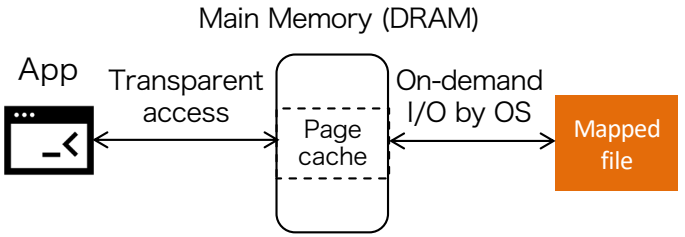
- Should leverage the file system
  - Tremendous amount of powerful technologies

Can we allocate data into file directly and store the data as is while providing transparent accesses to applications?

Background
Memory-mapped File Mechanism (mmap() system call)

- Maps a file into a process's virtual memory (VM) space
- Applications can access mapping area as if it were regular memory
- *Demand paging*
  - OS performs I/O on-demand by *page* granularity (e.g., 4 KB or 64 KB)
  - OS keeps cache in DRAM (*page cache*)
- Can map a file bigger than the DRAM capacity





Background
Memory-mapped File Mechanism (mmap() system call)

- Example

```
int fd = open("/mnt/ssd/file", O_RDWR); // Open a file


int size = 1024;
// Maps a file into main memory (1024 bytes)
int* array = (int*)mmap(NULL, size,
                        PROT_READ | PROT_WRITE,
                        MAP_SHARED, fd, 0);


array[0] = 10;

msync(array, size, MS_SYNC); // Flush dirty pages into the file
munmap(array, size); // Close the mapping
```

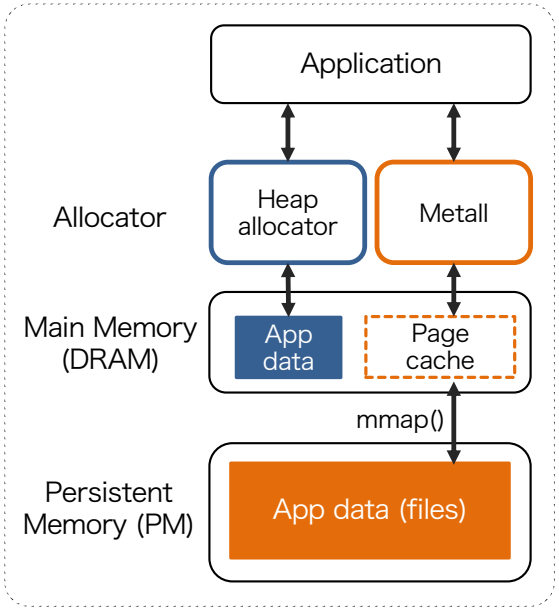
mmap is powerful; however,
calling mmap for every memory allocation is expensive

Metall[Iwabuchi'19]
A C++ Allocator for Persistent Memory

Fork me on GitHub

[github.com/LLNL/metall]

- A memory allocator built on top of a memory mapping region
  - Designed to work on any memory devices with file system support
  - Can leverage file system technologies
- Enables applications to allocate heap-based objects into PM, just like main-memory
- Can resume memory allocation work after restarting
- Incorporates state-of-the-art allocation algorithms
  - Some key ideas from SuperMalloc[Kuszmaul'15] and jemalloc
- Employs the API developed by Boost.Interprocess (BIP)
  - Useful for allocating C++ custom data structures in PM



Persistent Memory Allocation using Metall

Create new data (create.cpp)

```
void main () {
metall::manager metall_mgr(metall::create_only, "/ssd/test");
int* n = metall_mgr.construct<int>("val0")();
*n = 10;
}
```

Allocate a manager object

Directory to store data

Allocate and construct an object

Store a key to retrieve the data later

Metall::manager's destructor synchronies data with the PM (files)

Terminal

```
$ ./create
$ ./open
10
```

Reattach the data (open.cpp)

```
void main () {
metall::manager metall_mgr(metall::open_only, "/ssd/test");
int* n = metall_mgr.find<int>("val0").first;
std::cout << *n << std::endl;
}
```

Retrieve data with its key

Data is directly accessed in PM (no serialization overhead)

Metall with C++ Standard Template Library (STL) Container

A vector type with the STL allocator in Metall

```
using vec_t = vector<int, metall::allocator<int>>;
```

Template parameters of the STL vector container

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```

Create new data (create.cpp)

```
void main () {
metall::manager metall_mgr(metall::create_only, "/ssd/test");
vec_t* pvec = metall_mgr.construct<vec_t>("vec")(metall_mgr.get_allocator());
pvec->push_back(10);
}
```

← Can use it normally, including changing its capacity

↑ Arguments to vec_t's constructor

Reattach the data (open.cpp)

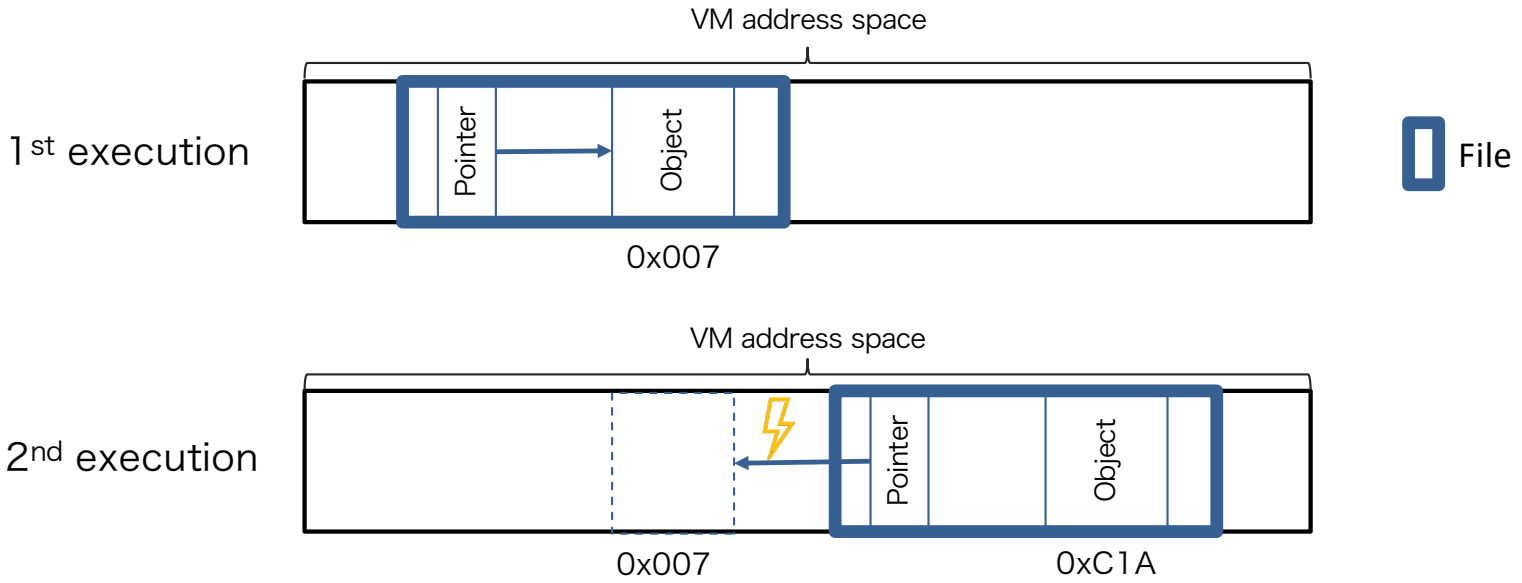
```
void main () {
metall::manager metall_mgr(metall::open_only, "/ssd/test");
auto pvec = metall_mgr.find<vec_t>("vec").first;
pvec->push_back(20);
}
```

← Can resume work, including changing its capacity

Metall follows the C++ standard style of using custom allocator
(no directives, no change to compilers)

Random Memory Placement

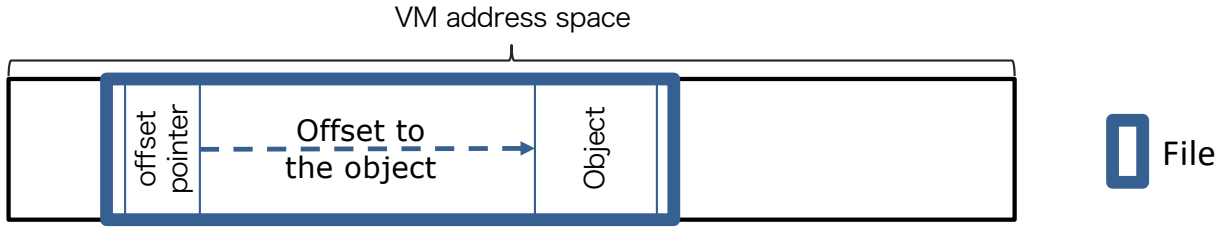
- Metall does not assume that file(s) are mapped to the same VM addresses every time



How to fix the random memory placement issue?

offset pointer

- An offset pointer holds the offset (distance) from itself to the object it points



  - Metall inherits offset_pointer implemented in Boost.Interprocess library
  - The concept of non-raw pointer is already integrated in C++

Possible implementation

```
template <class T>
struct offset_pointer {
  int64_t offset;
  ... many methods ...
}
```

Usage examples

```
struct data { int n };
data d;
offset_ptr<data> p(d);
p->n = 10;
p = nullptr;
```

```
int n[2];
offset_ptr<int> p(n);
p[0] = 1;
++p;
--p;
```

Solutions To Random Memory Placement

- Raw pointers
  - Must be replaced with *offset pointers*
- References, virtual function, and virtual base class
  - Must be removed since raw pointers are used
- STL Containers
  - Some implementations do not support offset pointers fully
  - Boost.Container library are compatible with Metall and Boost.Interprocess

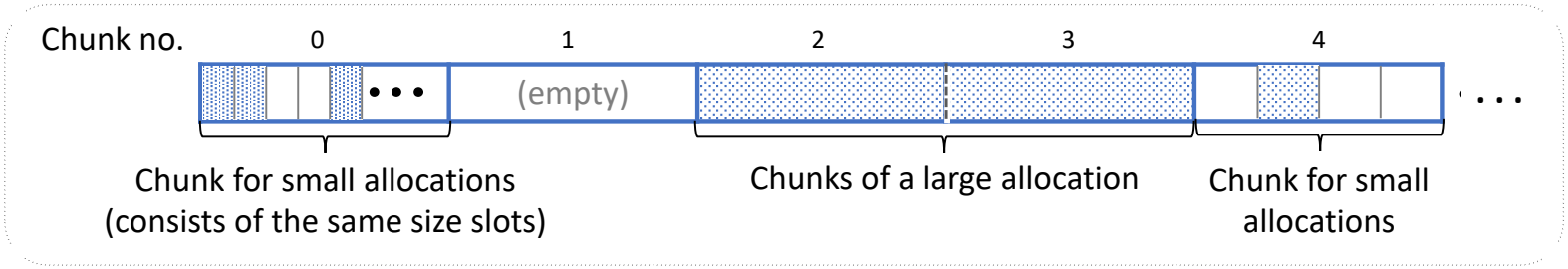
Metall Internal Architecture

Key design philosophy

- *Virtual memory is cheap in 64-bit machine, physical memory is dear* [Kuszmaul'15]
- Leverage *demand paging* (physical memory is not consumed until accessed)

-> Simplifies implementation & increases speed

- Application heap segment
  - Reserves a large continuous VM region in the process's address space
  - Maps backing files to the VM region on demand
    - Efficiently uses PM space and improve I/O performance



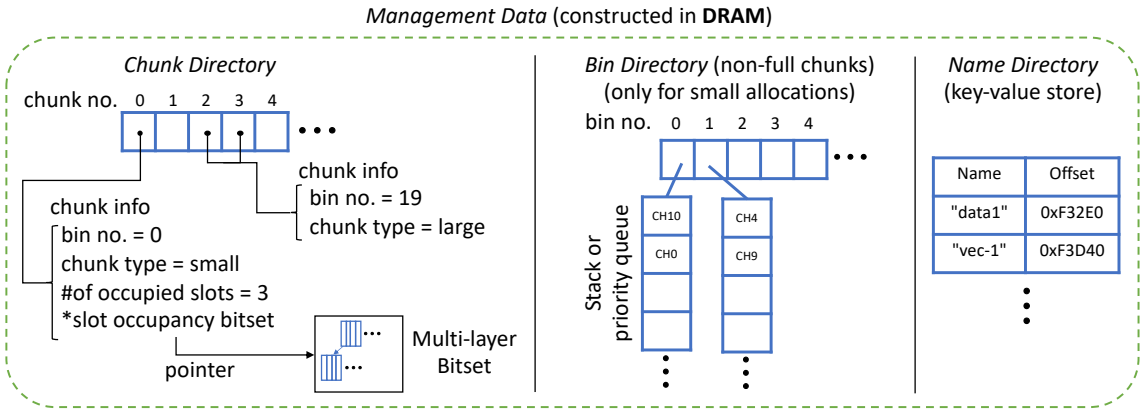
Allocation Size

- Small size category (e.g., <= 1 MB)
  - Rounded up to the nearest internal allocation size
  - Internal sizes are designed to keep internal fragmentations < 25%[Supermalloc][jemalloc]

- Large size category (e.g., > 1 MB)
  - Rounded up to the nearest power of 2
  - Designed not to waste physical memory much
    - uncommitted (non-touched) page does not consume physical memory
    - Worst cases:
      - 1.6% when allocating 1MB + 1 B with 4 KB page
      - 6.3% when allocating 1MB + 1 B with 64 KB page

Memory Allocation Management Data (SLUB allocator)

- Allocated in DRAM, separating from application heap segment to improve data locality
  - Unserialized/serialized when Metall's constructor/destructor is called



- Employs state-of-the-art allocation algorithms
- Free-slot caches
  - CPU core level to improve multi-thread performance

Snapshot/versioning in Metall

- Metall employs coarse-grained ('snapshot') persistence policy

```
metall::manger manager(...); // mmap()
// Application does some work:
// memory allocations and write operations
manager.~metall::manager(); // msync() and munmap()
```

Data is consistent

Data is inconsistent

Data is consistent

  - snapshot() creates a snapshot

```
metall::manger manager(...);
// Application does some work
manager.snapshot('/mnt/ssd2/data');
// Application does some work
manager.~metall::manager();
```

calls msync() and copies the mapped files to the '`/mnt/ssd/data`'
'`/mnt/ssd/data`' is consistent if snapshot() finishes correctly

How to implement a lightweight snapshot?

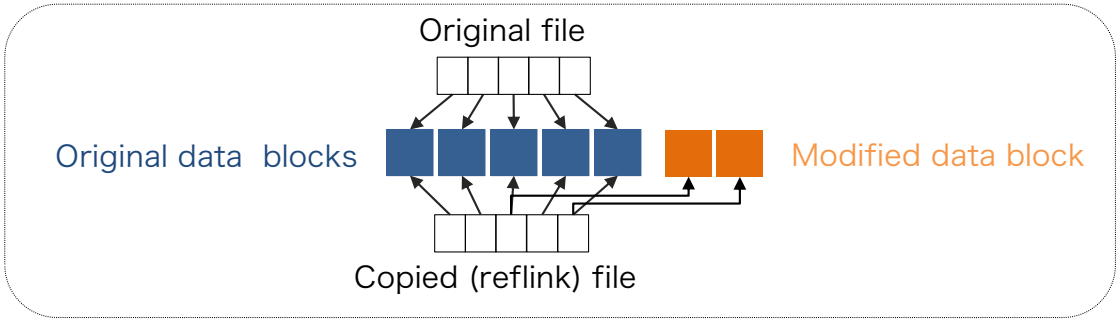
Persistence Policy — fine grained vs coarse grained

- Fine grained persistence policy
  - Synch data with persistent memory after every write operation
  - Ideal for transactional operations with recent byte-addressable PM
  - Can incur an unnecessary overhead for non-transactional apps

- Coarse grained persistence policy
  - Metall employs 'snapshot' consistency
  - A snapshot is created when the destructor or a snapshot method is invoked
  - Could cause data inconsistency if there is a clash before 'snapshot'

Lightweight Snapshot/versioning in Metall

- Calls msync() and copies backing-files to another location using *reflink*
- reflink
  - copy-on-write file copy mechanism implemented in filesystems (e.g., XFS, ZFS, Btrfs)

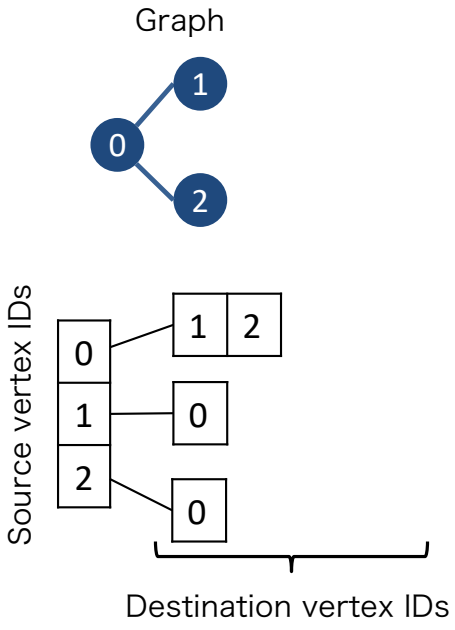


  - In case reflink is not supported by the filesystem, Metall automatically falls back to a regular copy

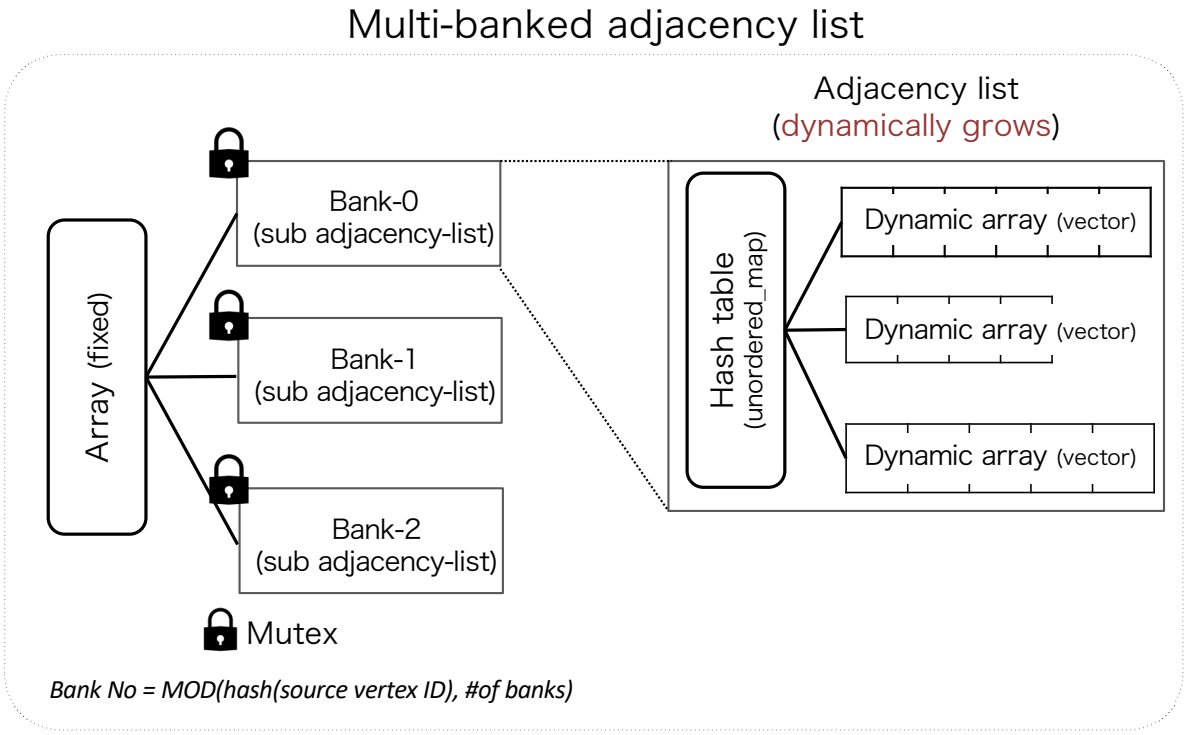
Lightweight snapshot is useful for many situations:
e.g., incremental data processing and crash consistency (node failure, application bugs)

Evaluation
Graph Data Structure

- Adjacency list (one of de-facto standard graph data structures)



Vertex ID is a 64-bit int



Evaluation
Machine Configuration

- Used two single-node machines at LLNL

EPYC (conventional PM device)

Storage	NVMe SSD
DRAM	256 GB
CPU	AMD EPYC CPU x 2 (96 threads)

Optane (byte addressable PM device)

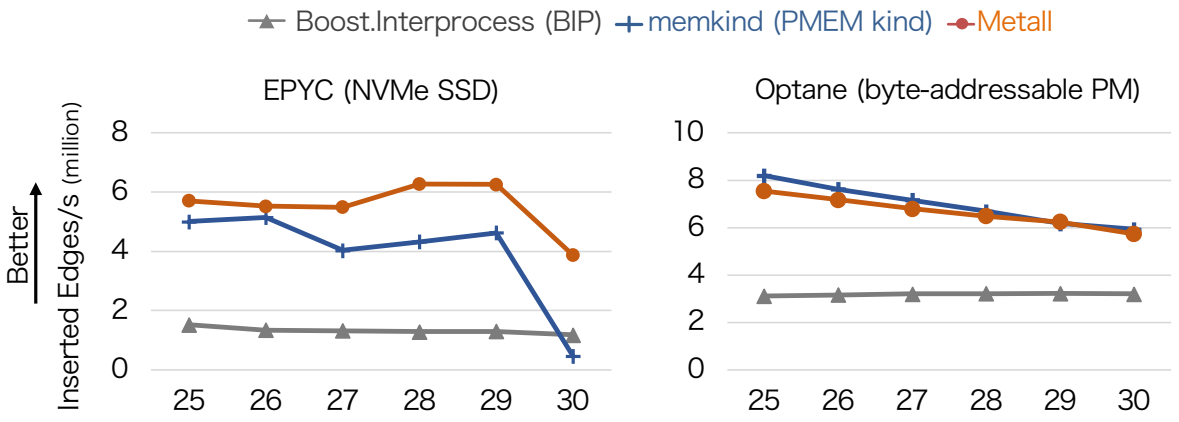
Storage	Intel Optane DC Persistent Memory (App Direct Mode + DAX)
DRAM	192 GB
CPU	Intel Skylake x 2 (96 threads)

Evaluation Result

Dynamic Graph Construction

shared-memory
multi-thread

- Baselines (memory allocators that use file-backed mmap underneath)
  - Boost.Interprocess
    - Uses a single tree structure for memory allocation management
  - memkind (PMEM kind)
    - Provides an allocator built on top of *jemalloc*
    - Cannot reattach data (uses PM as extended volatile memory)



Graph SCALE (the number of inserted edges = 2^{SCALE} x 32)

(SCALE 30 is larger than DRAM)

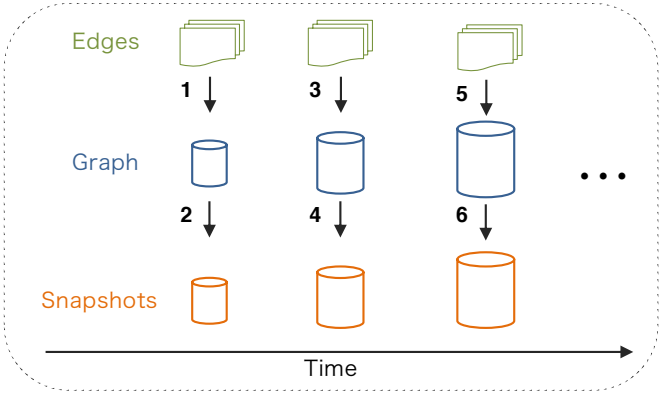
Metall provides persistent memory features whereas PMEM kind does not.

Evaluation Result

Incremental Graph Construction, Taking Snapshots

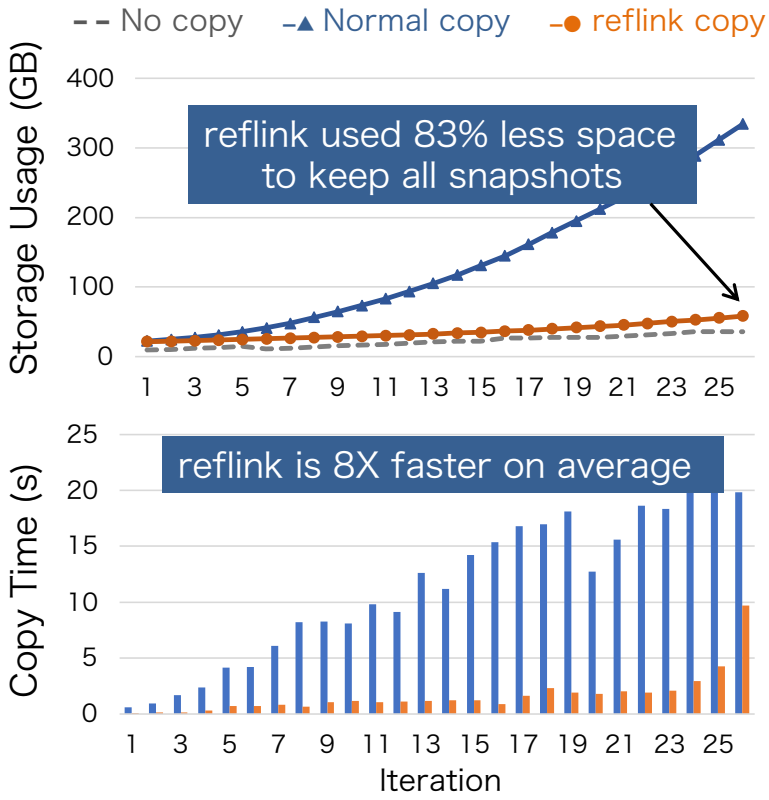
*reflink: copy-on-write file copy

- Workload
  - Take a snapshot after inserting each chunk (64M edges)
  - Insert edges into the original graph



- Dataset
  - Wikipedia page link insertions (1.8B edges)
    (curated by parsing English Wikipedia's revision history)

- Machine
  - EPYC machine (NVMe SSD with XFS filesystem)



Related Work

- Heap allocators (e.g., jemalloc, tcmalloc, malloc implementations)
  - Many studies have been conducted and showed notable results
  - Cannot persistently store their internal structures
- NVMalloc
  - Allocates memory on a distributed non-volatile memory (NVM) storage system
  - Creates a file per memory allocation request
- libpmemobj (in PMDK)
  - Employs a fine-grained persistence policy (ideal for transactional operations)
  - Requires fine-grained cache-line flushes to the persistent media
- Boost.Interprocess
  - Designed for interprocess communication (not designed as a persistent memory)
  - Employs a single tree structure to govern memory allocation

Summary of Metall

- Key features
  - Enables applications to allocate heap-based objects into PM, just like main-memory
  - Rich API for custom C++ data structures
  - State-of-the-art allocation algorithms
  - Efficient snapshost using *reflink*
- Open source
  - Available at github.com/LLNL/metall
  - Full documentation is hosted on Read the Docs
  - Travis CI and GitLab CI (in LC) are set up

CASC

Center for Applied
Scientific Computing

Lawrence Livermore
National Laboratory