Metall: An Allocator for Persistent Memory

(this work has been previously published in [1])

Keita Iwabuchi, Roger Pearce, and Maya Gokhale
Center for Applied Scientific Computing, Lawrence Livermore National Laboratory
Email: {kiwabuchi, rpearce, gokhale2}@llnl.gov

I. Introduction

Data-intensive applications play an essential role across many real-world data science domains. Often, these applications require storing data beyond a single process lifetime. Data often requires transformation into analytic-specific data structures to perform the analytic with reasonable execution time. The task of ingesting data, indexing and partitioning data in preparation of running an analytic, is often more expensive than the analytic itself. The promise of persistent memory is that, once constructed, data structures can be re-analyzed and updated beyond the lifetime of a single execution, and new forms of persistent memory are increasing the viability of processing complex data analytics.

We present Metall[1], a persistent memory allocator designed to provide developers with an API to allocate custom C++ data structures in both block-storage and byte-addressable persistent memories (e.g., NVMe SSD and Intel Optane DC Persistent Memory). Metall relies on a file-backed *mmap* mechanism to map a file in a filesystem into the virtual memory of an application, allowing the application to access the mapped region as if it were regular memory. Such *mmap* mappings can be larger than the physical main-memory of the system, allowing applications to address datasets larger than physical memory (often referred to as out-of-core or external memory).

Metall incorporates state-of-the-art allocation algorithms in SuperMalloc [2] with the rich C++ interface developed by Boost.Interprocess [3], and provides persistent memory snapshotting (versioning) capabilities.

We present a performance evaluation using a graph construction benchmark, persistently storing data structures on NVMe SSDs and byte-addressable Optane. Our approach using Metall works without code modifications in both sceneries — block-addressable and byte-addressable. Metall provides a coarse-grained consistency model, allowing the application to determine when it is appropriate to create durable snapshots of the persistent heap, and provides improved performance over Boost.Interprocess [3] and *memkind* [4] on an NVMe device.

II. Metall: Persistent Memory Allocator

1) API: Metall leverages the Boost.Interprocess (BIP) API to allocate objects into persistent memory, providing a means for custom C++ objects, including Standard Template Library (STL) containers, to be created dynamically and persist beyond the lifetime of the process creating them.

An example of storing and reattaching an STL container using Metall is shown in Code 1. There are some adaptations that must be performed to store objects in persistent memory. For example, raw pointers have to be replaced with *offset pointers* because there is no guarantee that backing-files are mapped to the same virtual memory address at each execution. An offset pointer holds the offset between the address of the referenced object and itself. The containers in Boost.Container library and libc++ work with offset pointers. Once an application has been adapted to have such containers/data structures, it is straightforward to use Metall.

Code 1: Example of using a STL container with Metall

```
using vec_t = vector<int, metall::allocator<int>>;
{
 metall::manager mgr(metall::create_only, "/ssd/mydata");
 //Allocate and construct an object of vec_t with key "vec"
 //Pass an STL—style allocator object to vec_t's constructor
 vec_t* pvec = mgr.construct<vec_t>("vec")
                              (mgr.get_allocator<int>());
 pvec—>push_back(5); // Can use the vector object normally
}
// — Exist the program and reattach the data — //
{
 metall::manager mgr(metall::open_only, "/ssd/mydata");
 vec_t* pvec = mgr.find<vec_t>("vec").first;
 pvec—>push_back(1); // Can change the data and capacity
}
```

2) Persistence Policy: Metall employs snapshot consistency, an explicit coarse-grained persistence policy in which persistence is guaranteed only when the heap is saved in a "snapshot" to the backing store. The snapshot is created when the destructor or a snapshot method in Metall is invoked. Those methods flush the application data and the internal management data in Metall to the backing store (backing files). If an application crashes before Metall's destructor finishes successfully, there is a possibility of inconsistency between the memory mapping and the backing files. To protect application data from this hazard, the application must duplicate the backing files before reattaching the data either through the snapshot method or else using a copy command in the system.

In contrast, libpmemobj in the Persistent Memory Development Kit (PMDK) [5] builds on Direct Access (DAX) and is designed to provide fine-grained persistence. Fine-grained persistence is highly useful (or almost necessary) to implement transactional object stores, leveraging new byte-addressable persistent memory fully, e.g., Intel Optane DC Persistent Memory. However, fine-grained persistence requires fine-grained cache-line flushes to the persistent media, which can incur an unnecessary overhead for applications that do not require such fine-grained consistency [6]. It is also not possible to efficiently support such fine-grained consistency on more traditional NVMe devices.

3) Snapshot: In addition to the allocation APIs, Metall provides a snapshot feature that stores only the difference from the previous snapshot point instead of duplicating the entire persistent heap by leveraging reflink [7].

[1]Metall is available at https://github.com/LLNL/metall

With reflink, a copied file shares the same data blocks with the existing file; data blocks are copied only when they are modified (copy-on-write). As reflink is relatively new, not all filesystems support it. Those that do include *XFS*, *ZFS*, *Btrfs*, and *Apple File System (APFS)* — we expect that more filesystems will implement this feature in the future. In case reflink is not supported by the underlying filesystem, Metall automatically falls back to a regular copy.

4) Internal Architecture: To efficiently manage memory allocations without a complex architecture, Metall uses Supermalloc's main design philosophy [2] – virtual memory (VM) space on a 64-bit machine is relatively cheap, but physical memory is dear.

Segment and Chunk: Metall *reserves* a large contiguous virtual memory (VM) space (e.g., a few TB) to map backing file(s) when its manager class is constructed. Metall divides the VM space into *chunks* (the default chunk size is 2 MB). Each chunk can hold multiple *small objects* (8B–half chunk size) of the same allocation size. Objects larger than the half chunk size (*large objects*) use a single or multiple contiguous chunks. Metall frees DRAM and file space by chunk.

Allocation Size: Metall rounds up a small object to the nearest internal allocation size as proposed by Supermalloc [2] and jemalloc [8]. Those allocation sizes are designed to keep internal fragmentations equal to or less than 25% and convert a small object size to the corresponding internal allocation size quickly. On the other hand, a large object is rounded up to the nearest power of 2. Although this strategy waste VM space, it does not waste physical memory on the unused pages.

Management Data: Metall uses three types of management data as follows to manage memory allocations: I) an array that holds the status of all chunks; II) lists of non-full chunk IDs to quickly find available space for small allocations; III) a simple key-value store used to store the keys and addresses of constructed objects. Metall constructs management data in DRAM to increase data locality; hence, Metall does not touch persistent memory when allocating memory. Metall deserializes/serializes the management data from/to files when its constructor/destructor or snapshot function is called.

III. Evaluation

We evaluate Metall by conducting graph construction because it is an example workload that is often more expensive than downstream analytics.

1) Setup: To construct graph data, we used one of de facto standard graph data structures, adjacency-list. Our adjacency-list data structure consists of an unordered_map (hash-table) and vector containers per vertex in the graph; it grows dynamically, requesting many memory allocations and deallocations. We implemented the data structure to take an allocator type in its template and an allocator object in its constructor like the containers in STL do so that it works with custom allocators.

We used two Linux (kernel v5) machines: the **EPYC** machine has dual sockets (96 threads), 256 GB DRAM, and a 1.6 TB of PCIe NVMe SSD (Ultrastar SN200 HH-HL add-in card) with XFS filesystem; the **Optane** machine has dual sockets (96 threads), 192 GB DRAM, and a 700 GB of Intel Optane DC Persistent Memory with App Direct Mode and ext4 DAX filesystem to bypass the page cache.

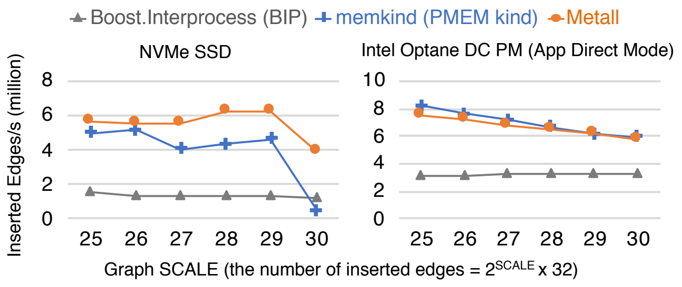

Fig. 1: Results of a dynamic graph construction benchmark using an R-MAT graph and two persistent memory technologies.

2) Dynamic Graph Construction: We conducted a multi-threaded dynamic graph construction benchmark on both machines. The benchmark inserts edges into an adjacency-list object allocated in a persistent memory device. We also used allocators in Boost.Interprocess (managed mapped file) and memkind library for reference. Although memkind uses persistent memory as volatile memory and cannot store data persistently, it provides a file-backed memory allocator (PMEM kind) built on top of a state-of-the-art heap allocator (jemalloc [8]).

We show the results in Figure 1. On the EPYC machine (NVMe SSD), Metall showed up to 4.3x and 8.3x improvements over BIP and memkind, respectively. We observed performance drops with memkind and Metall at SCALE 30 where the adjacently-list objects exceeded the DRAM capacity — even though Metall kept the best performance. On the Optane machine, Metall showed an almost identical performance to memkind and achieved 1.8–2.4x better performance over BIP.

Metall works without code modifications in both sceneries – block-addressable and byte-addressable, and provides improved performance over BIP and memkind on the NVMe device.

3) Snapshot with reflink: Metall provides a coarse-grained consistency model, allowing the application to determine when it is appropriate to create durable snapshots of the persistent heap. We also ran another incremental graph construction program, taking snapshots periodically, to show the efficiency of the snapshot capability implemented in Metall that uses reflink. We used a Wikipedia dataset; it contains 1.8 billion hyperlink insertions extracted from the revision history of English Wikipedia. We divided the dataset into 26 chunks and took a snapshot after inserting each chunk using normal copy or reflink copy. We used EPYC machine. When the last chunk was processed, reflink copy used 83% less storage space to store all snapshots compared with a full copy snapshot and showed 8x faster copy speed on average.

Acknowledgment

Prepared by LLNL under Contract DE-AC52-07NA27344 (LLNL-ABS-799306). Experiments were performed at the Livermore Computing facility. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

[1] K. Iwabuchi, L. Lebanoff, R. Pearce, and M. Gokhale, "Metall: A persistent memory allocator enabling graph processing," in *IA3 Workshop*, https://www.osti.gov/servlets/purl/1576900, 2019.
[2] B. C. Kuszmaul, "SuperMalloc: A super fast multithreaded malloc for 64-bit machines," in *ISMM '15*. ACM, 2015, pp. 41–55.
[3] "Boost Libraries," https://www.boost.org.
[4] "memkind," http://github.com/memkind/memkind.
[5] "pmem.io persistent memory programming," https://pmem.io.
[6] S. Haria, M. D. Hill, and M. M. Swift, "MOD: Minimally ordered durable datastructures for persistent memory," 2019.
[7] "ioctl_ficlonerange(2) - linux manual page," http://man7.org/linux/man-pages/man2/ioctl_ficlonerange.2.html.
[8] J. Evans, "A scalable concurrent malloc (3) implementation for FreeBSD," in *Proc. of the bsdcan conference, ottawa, canada*, 2006.