The paper describes the development of Metall, a persistent memory allocator, and its performance improvements compared to other memory management systems. The paragraph is well organized and contains technical details, but could benefit from some editing to make it more concise and easier to understand for a non-technical audience. Additionally, the comparison to Boost. Interprocess and memkind (PMEM kind) could be made more clear, and the paragraph could benefit from a concise explanation of what persistent memory is.

The introduction provides an overview of the importance of data analytics and the need for persistently storing data beyond a single process lifecycle. It then introduces Metall, a persistent memory allocator, and highlights its features and benefits. The introduction provides a good overview of the topic and the contributions of the work, but could benefit from some editing to make it more concise and easier to understand for a non-technical audience. The technical terms and references to previous works could also be explained in more detail to provide a clearer context for the reader.

This section provides a clear overview of the technology behind the use of NVRAM for data storage. It highlights the performance improvements and cost reductions in NVRAM technology and mentions the various types of NVRAM devices in production systems. The explanation of why a file system is highly beneficial for storing data in NVRAM is concise and easy to understand. The definition of persistent memory is clearly stated and provides a comprehensive understanding of what the term represents in the context of this paper. Overall, the section is well-written and provides a solid foundation for the rest of the paper.

This paragraph describes the use of the memory-mapped file mechanism to avoid the cost of data serialization in terms of performance and programming cost. The mmap system call is used to map a file into the process's virtual memory space and provide applications with transparent access to the region. The actual I/O is performed with the demand paging mechanism and is performed on-demand by page granularity. The mmap system call is essential for memory management but direct calls for each memory allocation can cause significant overhead. To mitigate this, another memory allocation management layer can be built on top of the memory-mapping region. The paragraph provides a code example of using mmap to map a file and explains the process of I/O. The content is well-structured and easy to follow.

It is easy to understand that Metall serves as a memory allocator for applications, utilizing the memory-mapped file mechanism and supporting persistence and snapshots. The explanation of its use of internal data and limitation to single process use is also clear. Overall, the description effectively conveys the key features of Metall.

We then have a clear description of the C++ interfaces provided by Metall for persistent memory allocation, and how it leverages the APIs from Boost.Interprocess library. The use of familiar memory allocation functions such as allocate and deallocate and constructors like construct help in its usability for applications. The description of the principal APIs in a table could be helpful for developers using Metall. However, it would be useful to mention the level of abstraction provided by these interfaces and how they hide the details of mmap system calls from the users.

We then are provided a comparison of the persistence policy employed by Metall with that of another library, libpmemobj in the PMDK. It highlights the benefits and limitations of the snapshot consistency used by Metall and the fine-grained persistence used by libpmemobj. The writer explains that the fine-grained persistence provided by libpmemobj can incur unnecessary overhead for some applications, but is necessary for others. The explanation is clear and provides enough detail for the reader to understand the difference between the two policies.

The description of Metall's snapshot feature is clear and concise, including its use of reflink for optimization and its fallback mechanism for unsupported filesystems. However, it would be useful to mention the benefits of using reflink, such as improved storage efficiency and speed, as well as any trade-offs, such as limited availability and potentially less compatibility with certain file systems. Additionally, it would be helpful to explain how the snapshot feature works in more detail, such as what exactly is stored in the difference and how it is used in subsequent runs of the application.

The description of this paragraph provides a clear explanation of why raw pointers cannot be used in persistent memory allocation and the need for offset pointers. The use of offset\_pointer in Metall and the compatibility of STL containers with Metall are also highlighted. The mention of other persistent memory allocators that use similar designs to offset pointers and the integration of non-raw pointers into C++ adds context and supports the explanation. However, the limitations and restrictions of references, virtual functions, and virtual base classes in persistent memory could have been more clearly explained. The suggestion of developing a program to assess the compatibility of data structures with Metall is a good idea for future work.

This passage describes the design of the Metall system for storing and managing data in persistent memory. It mentions the creation of a root directory for storing the management data and application data allocated through the manager object. The fact that all files related to a single manager are in the same directory makes it easy to duplicate or delete a Metall datastore using normal file copy or remove commands. It also mentions that Metall is not designed for multi-process data sharing, but multiple processes can still open the same datastore in read-only mode. Additionally, the system uses multiple files to store application data, which can improve parallel I/O performance, and the files are created and mapped on demand, with a default file size of 256 MB.

Overall, the passage provides clear and concise information on the design and features of the Metall system. However, it would be useful to include a mention of the potential benefits and limitations of using this system compared to other solutions for managing data in persistent memory.

The description of the virtual memory reservation by Metall seems clear and well-explained. It's good to mention the default value and the possibility of customizing the reservation size and chunk size. The mechanism of dividing the reserved memory into chunks for different object sizes and freeing memory by chunk also seems well-explained. However, it would be helpful to mention the trade-off between reserving more virtual memory and the corresponding impact on physical memory usage.

The passage describes how Metall handles memory allocation for both small and large objects. For small objects, Metall uses rounding up to the nearest internal allocation size, following allocation sizes proposed by Supermalloc and jemalloc. This approach helps to keep internal fragmentation at a minimum and improve allocation efficiency. For large objects, Metall rounds up to the nearest power of 2. Although this strategy can result in wasted virtual memory space, it does not waste physical memory due to the demand paging mechanism. The author mentions that in the worst case, physical memory waste can be 1.6% and 6.3% for 4 kB and 64 kB page size systems, respectively.

This description of Metall's memory management system seems clear and concise, but it could benefit from some elaboration on the specific details of each of the three management data directories. What exactly is stored in each directory and what purpose does it serve? Additionally, a little more information on the serialization/deserialization process, such as why it is necessary, how it is performed, and how it affects performance, would be useful for the reader's understanding.

This passage describes the chunk directory, one of the three management data directories used in Metall to manage memory allocation. It mentions that the chunk directory is an array of blocks that hold information about the status of the chunks, such as the bin number, type of chunk, and a pointer to a bit set. The use of a compact multi-layer bitset table and built-in bit operation functions is described as a way to efficiently manage available slots in a chunk for small allocations. The maximum number of slots that can be managed is (2^18). The passage mentions that sequential probing is used to find empty chunks and that the performance can be improved by implementing an additional index structure.

The explanation of the bin directory is clear and concise. It accurately describes the purpose and behavior of the bin directory. The use of the LIFO manner is appropriate for small allocations. Mentioning the reason for not using the bin directory for large allocations is also helpful for understanding the design decisions. Overall, this section provides a good understanding of the bin directory's role in memory allocation management.

This explanation of the name directory is clear and concise. However, it would be helpful to have more information on why the name directory is important and how it is used in conjunction with the other two management data directories (chunk and bin). Additionally, it would be useful to have information on the performance characteristics of the name directory and how it affects the overall performance of Metall.

This explanation of the use of an offset pointer in Metall STL allocator is clear and concise. The reason for using the offset pointer is well justified and the explanation of how it works is easy to understand. The method described seems to be a viable solution to the issue of accessing management data from the allocator object.

The use of a single mutex object for the chunk directory and the name directory each seems reasonable, but the use of a mutex object per bin in the bin directory might result in increased overhead due to the need to lock and unlock multiple mutex objects. This can lead to decreased performance if multiple threads are accessing the bins concurrently. The potential bottlenecks described in the two situations might also impact the performance, so it's important to consider alternative solutions to address these potential performance issues.

The use of free-object caches at the CPU core level is a good design decision for increasing multi-thread performance, as it takes advantage of the cache hierarchy and reduces cache-coherence traffic. However, a potential drawback of this design choice is that it may limit scalability, as the number of cache objects is fixed at the number of CPU cores and does not scale up with increasing number of threads or CPU sockets. It may be worth considering implementing additional levels of cache hierarchy (such as thread or CPU socket level) to further increase performance and scalability.

This description provides an overview of the problem with using mmap with network file systems and presents bs-mmap as a solution. It mentions the limitations of a naive solution (data staging) and the drawbacks of tuning the page cache. The explanation of bs-mmap is clear, but it would be helpful to have more details on the implementation and how it addresses the issues with mmap and network file systems. Additionally, the explanation of the benefits of using bs-mmap over other solutions would also be useful.

This description of the bs-mmap implementation seems to be technically accurate. The use of the /proc file system and the pagemap file to identify dirty pages in a private mapping is an interesting approach. However, it is important to consider the trade-offs and limitations of using this approach compared to other options, such as using msync with a shared mapping or using a different mechanism for page tracking. Additionally, it would be useful to provide some context around the performance and efficiency benefits of using bs-mmap, and to compare it with related solutions.

This passage describes two optimizations made in the design of bs-mmap to improve performance on parallel file systems. Writing back dirty pages in consecutive chunks instead of page-by-page is a good optimization to reduce the overhead of frequent small I/O operations, as it allows the file system to handle larger contiguous I/O requests. The parallel I/O approach, where bs-mmap assigns a separate thread to each file, is also a good optimization to take advantage of the parallelism offered by modern file systems. However, it is important to consider the potential trade-offs that come with parallel I/O, such as increased complexity and the risk of creating contention or race conditions.

The description of the implementation of the multi-bank adjacency list for graph construction with multiple threads on a shared-memory system appears to be clear and concise. It explains the use of the unordered\_map and vector containers for the vertex table and edge list, respectively, and how they use a hash table and dynamic array to efficiently store the graph data. The use of m banks and associated mutex objects to support multi-thread graph construction is also well explained. The customization of the multi-bank adjacency list to work with a custom STL allocator by using an allocator-aware class is a good implementation choice and is described in sufficient detail.

The authors describe the use of three single-node machines at Lawrence Livermore National Laboratory for their experiments. The first machine, named "EPYC," has a PCIe NVMe SSD and the page cache behavior was optimized for improved performance. The second machine, named "Optane," has an Intel Optane DC Persistent Memory device installed and configured with the App Direct Mode and ext4 filesystem DAX mode. The third machine, named "Corona," is one of the nodes in a cluster of over 200 compute nodes, connected to two parallel file systems, Lustre and VAST, which have different performance characteristics for different types of I/O. The authors provide specifications for each machine.

This passage describes the different persistent memory allocators that were used in the study. The author explains the features of Boost.Interprocess, PMEM kind, and Ralloc, and mentions the libraries and versions used. The author also mentions a change made to PMEM kind for better performance on the Optane machine, and why Ralloc was only used on the Optane machine. The author concludes by mentioning that a custom STL-compatible allocator class was written using Ralloc.

In general, the passage is clear and concise, although it may benefit from some reorganization to better explain the relationships between the different allocators and the purpose of using each one. It may also benefit from some elaboration on the specifics of how each allocator works and how it was implemented in the study.

This passage describes the process of generating synthetic scale-free graphs for benchmarking purposes using an R-MAT generator. The graphs generated range in size from SCALE 24 to 30 and the vertex IDs are scrambled to remove unexpected localities. The edges in the graph are treated as undirected and the number of actual edges inserted is calculated based on the formula given. The edge generation is performed in chunks and the time taken to generate the edges is excluded from the reports.

This passage reports the results of a benchmark comparing the performance of three different graph implementations: Metall, Boost.Interprocess (BIP), and PMEM kind, on two different machines: the EPYC and the Optane machine.

For the EPYC machine, the results show that Metall significantly outperforms BIP and PMEM kind, with improvement factors ranging from 7.4 to 10.9x and 2.2 to 2.8x, respectively, at different graph sizes. However, at the largest graph size (SCALE 30), there is a drop in performance for all implementations due to DRAM capacity limitations. At this scale, Metall still outperforms BIP and PMEM kind by 11.7x and 48.3x, respectively.

For the Optane machine, the results show that Metall has slightly better performance than BIP, with improvement factors of 2.1 to 2.3x. The results also show that Ralloc did not finish processing the largest graph size due to a lack of persistent memory space. The performance of Metall was comparable to that of Ralloc and the modified PMEM kind, with PMEM kind and Ralloc being up to 10% and 14% better than Metall, respectively.

This explanation of the performance difference between Boost.Interprocess and Metall is well-reasoned and clear. It gives a clear explanation of the performance limitations of Boost.Interprocess and why Metall was able to achieve better performance. The comparison of Metall's performance on EPYC and Optane machines highlights its high portability. The reference to the design strategies from Supermalloc gives credibility to the explanation.

The passage describes an experiment that evaluates the performance of constructing a persistent graph data structure using the Metall tool on two different network file systems (Lustre and VAST). The authors use real temporal graph datasets from Wikipedia and Reddit and sort the edges by timestamp to simulate a growing graph. The benchmark involves constructing a banked adjacency list incrementally by partitioning the sorted datasets into monthly chunks and adding each chunk to a Metall data store, measuring the time per iteration. The measured time is broken down into ingestion time and flush time. The experiment was run on a single node of the Corona cluster.

The paper describes two datasets used for the experiments. The first dataset is the Wikipedia page reference graph which was curated by extracting hyperlinks between all pages in the English Wikipedia dump as of July 1st, 2017. The graph contains 1.8 billion hyperlink insertions and includes not just article pages but also author and category pages. The second dataset is the Reddit author-author graph, which was constructed by extracting user activities from the Reddit social news website. This dataset represents comment interactions as edges from one user to another and contains 4.4 billion comment activities.

The paper describes three configurations for mapping files into Metall's virtual memory space during the graph construction process. The first configuration, "direct-mmap," maps files directly from Lustre or VAST into Metall's virtual memory. This is considered the baseline for performance comparison. The second configuration, "staging-mmap," uses tmpfs as a temporary file storage and maps files from there into the virtual memory. The staging step involves copying the datastore from Lustre or VAST into tmpfs at the beginning and end of each iteration, with parallel file copy-in and copy-out operations for maximizing resources utilization. The third configuration, "bs-mmap," uses the MAP\_POPULATE flag to read the mapped file ahead into virtual memory and disables the feature of freeing file space in Metall for consistency across both file systems.

The authors of the study conducted experiments on two different network file systems, Lustre and VAST, to evaluate the performance of constructing graphs using the incremental benchmark with Metall. The authors evaluated three different configurations of mapping files, including direct-mmap, staging-mmap, and bs-mmap. The results showed that, on Lustre, staging-mmap showed the best performance for both Wikipedia and Reddit graphs, while on VAST, bs-mmap had the best performance. The authors concluded that the performance difference is due to the different bandwidth and file metadata access cost of the file systems.

The text describes the benefits of using graph analytics in data processing, specifically mentioning the challenge of persistently storing and updating data for analysis beyond a single execution. The author mentions GraphBLAS as a solution to this problem, which specifies building blocks for computing on graph-structured data using linear algebra, represented as sparse matrices and operations using an extended algebra of semirings. The author also mentions the GraphBLAS Template Library as a C++ implementation of the GraphBLAS specification.

The description of the typical workflow of GBTL seems clear and concise, but it would be helpful to mention the limitations of using a normal/transient memory allocator. Specifically, mentioning that the graph construction step must be repeated every time a graph algorithm is run may be relevant to the reader, as it highlights one of the drawbacks of this approach.

This section describes the implementation of GBTL with Metall to make the graph analytics data structures persistent. The authors have adopted the adjacency list data structure for the vertex and edge lists, and modified it to use a custom STL allocator instead of the default one. They have also modified the high-level graph algorithms in GBTL to take a graph type with a custom allocator. The authors have implemented a fallback allocator adaptor that can be used to quickly integrate Metall into an application. The only changes required to use Metall with GBTL were in the graph data structures and a few helper functions. The authors show how the original graph analytics code has to be changed to use Metall, specifically using Metall to allocate the graph and adding a graph reattach mode. The graph reattach mode would be useful for many graph analytics applications as it avoids the construction time.

The authors demonstrate the benefits of using the Metall persistent memory allocator by integrating it with GBTL, a breadth-first search and page rank implementation, and comparing its performance with the original GBTL implementation. Four datasets from SNAP were used, containing 1K to 7K vertices and 14K to 104K edges. The results showed that GBTL+Metall achieved 3.5X better BFS execution time and similar improvements in page rank execution time compared to Base GBTL, mainly due to the ability to avoid heavy graph construction time. The authors conclude that integrating Metall in GBTL enables interactive real-time data science applications with large persistent data structures, improving software productivity.