Asynchronous Logging and Fast Recovery for a Large-Scale Distributed In-Memory Storage

Kevin Beineke, Florian Klein,
Michael Schöttner
Institut für Informatik, Heinrich-Heine-Universität Düsseldorf,
Universitätsstr. 1, 40225 Düsseldorf, Germany
E-Mail: Kevin.Beineke@uni-duesseldorf.de

Abstract: Large-scale interactive applications and online graph analytic processing require very fast data access to many small data objects. DXRAM addresses these challenges by keeping all data always in memory of potentially many nodes aggregated in a data center. Data loss in case of node failures is prevented by an asynchronous logging on flash disks. In this paper we present the architecture of a novel logging service designed to support billions of small data objects, flash disk characteristics and fast node recovery. The latter is challenging if 32-64 GB of in-memory data of a failed node needs to be recovered in seconds from the logs.

1 Introduction

Large-scale interactive applications and online graph computations must often manage many billions of small data objects. Facebook for example supports more than one billion users by keeping around 150 TB of data in more than 1,000 memcached servers [ORS+11]. Around 70% of these objects are smaller than 64 byte [NFG+13] and as a result each memcached server stores a few hundred million of objects. The sheer amount of objects and the small data sizes can be found in many other online graph applications, e.g. [SWW+12]. Another important aspect are the access patterns of these applications where reads dominate over writes [BAC+13], [AXF+12]. And depending on the type of data there are also deletes and updates but both less frequent than reads.

DXRAM is a distributed in-memory system aiming at managing billions of small data objects [KS13]. The core implements a key-value data model for binary data with basic get and put operations. The system design is open to execute algorithms on storage nodes or to run DXRAM as an ultra-fast back-end storage. A super-peer overlay is used for scalable node lookup and a paging-like translation scheme for efficiently mapping global unique keys to local virtual addresses. On top of the core, we plan extended data services, e.g. richer data models and naming/indexing services.
In this paper we present the design of our novel asynchronous logging approach, which is storing replicas of objects on remote flash disks to prevent data loss potentially caused by software and node failures. Replication in memory is not an option as memory is very expensive and power outages may draw down many nodes resulting in data losses. The

logging is designed carefully, taking into account SSD characteristics, to maximize write performance. The latter requires basically sequential writes, which results in a sequential log. This however requires to provide a cleaning mechanism similar to the work published in log-structured file systems [RO92].
Another aspect is that we design the log in a way to allow very fast recovery of nodes states (32-64 GB of data), which is very challenging [ORS+11]. Fast recovery will be even more challenging as we aim at supporting billions of small data objects.

This paper is structured as follows. In section 2 we give a short overview of the DXRAM system including the global meta-data and in-memory data management. Section 3 presents the logging approach, followed by section 4 and 5 where we describe the recovery process and the cleaning of logs. Related work is discussed in section 6 followed by the conclusions and an outlook on future work.

2 DXRAM Architecture Overview

DXRAM is a distributed in-memory storage system designed for the management of binary data in data center environments. We aim at supporting billions of small data objects (16-64 byte) which is for example required for managing very large graphs, e.g. social networks. By keeping all data always in RAM we can provide low-latency data access for all data. Furthermore, by providing transparent persistence we relieve programers from synchronizing caches with secondary storage.

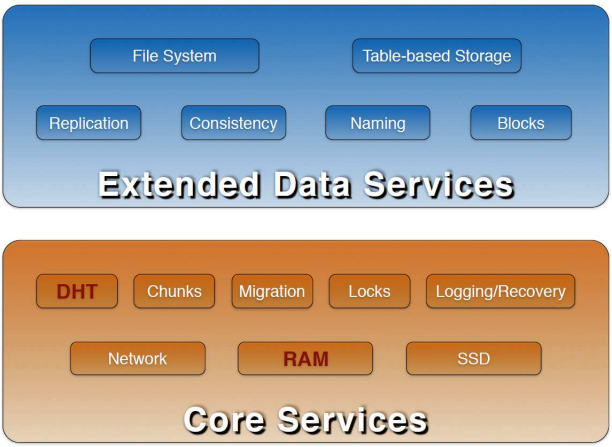

Figure 1: DXRAM architecture

Figure 1 shows the overall architecture of DXRAM. Extended data services include general services and extended data-models built upon core services. Not all extended services shown in the figure are implemented but we designed the core to be flexible allowing for example to support different data models as needed without requiring to re-implement

everything from scratch.

Core services provide functionality for the management, storage and transfer of chunks (key-value tuples). One of the main objectives in core services is to keep the functionality and the interface for high layers as compact as possible. The minimal interface for chunks includes following functions *create*, *delete*, *get*, *put* and *lock*.

The DXRAM core implements a key-value data model where a tuple is called chunk. A chunk has a 64 bit globally unique chunk ID (CID) followed by the value (raw bytes). The CID is composed of two parts. The first 16 bit is the node ID of the creating node (NID_C). The remaining 48 bit is a locally unique value, named LID, which is incremented during each local chunk creation. This results in a sequential CID generation scheme on every node. The key size is configurable but with the described numbers we can address 65,536 nodes each storing up to 280 trillion chunks. We decided to use this scheme in favor of user-defined keys to avoid hashing allowing us to keep IDs compact, to support locality-based access patterns and to avoid adjusting hash functions in case of nodes scaling up and down.
But the sequential ID generation is not a constraint for the applications. On the one hand most applications which use databases as persistent storage access data through auto-incremented row IDs similar to our LID. On the other hand DXRAM provides a name service to address a CID using a user-defined key. The super-peers store the mapping of keys and CIDs in a patricia-trie structure. The intention is that not each single object needs a user-defined key but only a subset, e.g. the user records in a social network.

Chunks have variable sizes defined during their creation always stored en bloc. The basic get and put operations on chunks read and update always full chunks. Because RAM is expensive we replicate chunks for fault tolerance on multiple backup nodes only in flash memory.

2.1 Global Meta-Data Management

A super-peer overlay is used for implementing a custom DHT allowing fast node lookup of chunks. Because of the high-speed network and the limited number of super-peers, e.g. 8-10% of all nodes we decided to keep them knowing each other in contrast to traditional internet-scale DHTs like Chord, CAN, etc. This in turn allows lookups with O(1) time complexity. In addition peers cache node-lookup results reducing load on super-peers. Meta-data is replicated on the neighboring super-peers to mask super-peer failures. However, we do not need a high replication factor, which would be very costly, as meta-data can be dynamically re-constructed.
In a controlled environment like a data center we expect only small node churn caused by failures. Of course a power outage may kill all our nodes and will require to restore all data from all logs, which will take considerable recovery time; but we expect this disaster to be very seldom. Otherwise we expect controlled up and down scaling of nodes as needed dynamically using the super-peer overlay. If we reach a defined threshold a new super-peer is promoted or respectively demoted. In both cases the hashing function for the DHT does

not need to be adjusted thus not requiring to re-hash entries (like for a traditional DHT).

The super-peer overlay can aggregate the meta-data of chunks because of the sequential CID creation and bundles multiple CIDs to a CID Range (start, end) together with the NID_O (o: actual owner). Note, the actual owner of an object may be different to the creator if an object has been migrated, e.g. because of load-balancing reasons.
For example, if we have 1,024 chunks with CIDs 1 to 1024 created on one node, then we have only one entry for this CID range on the associated super-peer. In addition to avoid gaps in the CID ranges through deletions of chunks, we reuse free CIDs for new chunks. As super-peers have to map billions of CIDs from all their associated peers this compact representation is very space efficient.

As mentioned above DXRAM also supports chunk migrations to relieve load on peers storing possibly several hot spots. For example in the context of social networks a hot spot can be a famous artist whose profile is very often visited (some artists have up to 40 million friends in Facebook [SWL13]). Although migration solves the load problem it results in a range split of the global ID range on the super-peer responsible for the migrated object. Still this is not a problem for the global meta-data management because we expect that such chunk migrations are seldom as we do not expect billions of hot spots. A fast lookup of CIDs in chunk ranges is ensured by implementing a B-tree-like data structured described in [KS13].

Beside meta-data management, super-peers are also responsible for recovery coordination of failed nodes which is discussed in Section 4 and for load monitoring which is beyond the scope of this paper.

2.2 In-memory Data Management

We aim at managing up to one billion of small objects on one node, which is challenging especially concerning the mapping of global IDs to local virtual addresses as well as the local memory management. Although both topics are well studied in literature the sheer amount of objects raise new challenges regarding space efficiency and access times.

Many systems use hash tables for the translation of global IDs to local addresses which tend to waste memory or get slow when the load factor increases (causing collisions). In contrast we implemented a paging-like translation scheme which is fast and space efficient.

Furthermore, most memory allocators are not optimized for many small allocations. On the one hand they align memory for cache performance reasons (4/8 byte granularity) and on the other hand they append a header (4 - 64 byte) depending on programming language and runtime system. Obviously, the overhead is too large, e.g. for one billion objects, each with a 16 byte header, we would need 16 GB RAM just for the headers. Because memory is expensive we have decided to design a memory management trading cache performance for minimizing internal fragmentation. Cache penalties are not critical for DXRAM (when used as back-end storage service) as the time for client operation requests is dominated by network transmissions.

The global ID mapping and the local memory management are described in a paper, currently submitted to another conference.

3 Organization of the Log

As mentioned before DXRAM keeps all data always in memory. In order to address node failures, power outages and not to burden the programmer with persistence management on secondary storage we introduce an asynchronous logging service described in this section. Each object is replicated on a configurable number of remote nodes called backup nodes. These backups are stored in a log optimized for SSDs (, which could also be applied to traditional disks). During fault-free execution only write operations of updates are executed on the log but no reads. In case of a recovery request, when a node crashed, the full log is read (described later in more detail). A fast recovery is only possible if the state of one node is spread over many backup nodes. This allows to aggregate SSD and network bandwidths during recovery [ORS+11]. Therefore, we introduce backup zones, similar to the segments in RAMcloud, although we do not use static sizes but allow dynamic size adaption. Each node providing RAM is also providing SSD backup-capacity.

Before we start with the presentation of the log service we discuss the characteristics of SSDs. In contrast to traditional disks, SSDs read and write clustered pages, each 4 KB and each on another NAND flash memory accessed via multiple channels. This means, writing one byte results in writing at least 4 KB. Obviously, as we aim at supporting many small data objects (16 - 64 byte), we need to introduce some buffering for bundling write accesses. To benefit from internal parallelism of SSDs it is even better to write multiple pages at once [MKC+12]. Furthermore, SSDs cannot re-write a page, but instead have to delete the block (64 to 128 pages) first. The write access will then be done on a page the SSD controller chooses. To manage the complexity of that fact the SSD controller uses a specific garbage collection and flash translation layer to hide the characteristics of flash memory. This results in a much higher bandwidth while writing sequentially than randomly [MKC+12]. Finally, two more aspects to be taken into consideration are to avoid delete operations whenever possible because of the large erase block granularity (depending on model, e.g. 256 KB - 2,048 KB) and not to mix read and write accesses as both can harm each other [CKZ09]. The latter is no problem as read and write operations are anyway separated (fault-free execution and recovery), see above.

3.1 Primary Log

DXRAM organizes logs into two levels, one primary log and several secondary logs, one for each node requesting backups. The objective of the primary log is to store incoming backup requests as soon as possible on SSD to avoid data loss caused by software or hardware failures. The primary log is organized as a ring buffer similar to SpriteFS' [RO92] and all incoming backup requests (from any node) are bundled together in the primary log.

In contrast to log-structured file-systems the log is never read during fault-free execution. Thus there is no need to store meta-data of the log in memory. However, the log is self describing, each entry has a header with a length field, NID and LID for identifying the object and optionally a CRC checksum, and is fully read-in during node recovery.

In order to avoid many small write accesses to the primary log, all incoming backup requests are bundled in a write buffer. The buffer itself is organized as a ring buffer and the access is performed using several producer and one consumer thread. The producer threads decouple network threads processing backup requests, however, the node requesting a backup may force a synchronous operation, if requested by the above application. This is optional in case of critical updates, under control by the applications.

It is important to note that the buffer will be filled by backup requests from potentially many nodes, which partially backup their state on a backup node. So, it is likely that there will soon be enough data in the buffer to flush one page out to the primary log or even multiples of a page, depending on the update frequency in the overall system. However, we set a configurable timeout ($<$1s) for the consumer thread to flush out data to SSD. The idea is to avoid an increased data-loss probability in case of very low update frequency.

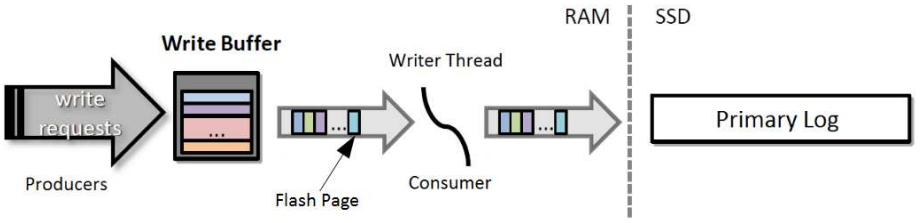

Figure 2: Interaction between write buffer and primary log

Obviously, there is a small time interval where an update is vulnerable to a power outage or failure of all nodes storing the update in memory. Of course in the worst case scenario the full data center could go down by a power outage. The latter is very unlikely, whereas the first may happen. As a consequence we provide applications a sync command to enforce write through for backups of critical updates.

Right now we would have to read in and analyze the full primary log during a recovery requests to detect log entries of the crashed node and skip all other entries. This is obviously inefficient, which in turn lead us to establishing the secondary log.

3.2 Secondary Logs

Each node distributes its state over numerous backup nodes, e.g. 64 GB RAM of one node is distributed over 100 or more backup nodes. Thus each node will service as storage node and as a backup node for potentially many other nodes. To speed up recovery we decided to sort backups into one separate secondary log for each node. Thus we can avoid reading in one large log storing backups from many nodes and analyzing which entries are needed

for a recovery.

After having written a backup request into the primary log, the data is also copied (in memory) into the secondary log buffer using the NID, see Figure 3. The secondary log buffer array grows dynamically to provide one write buffer for each node whose backups are to be stored. The reason for buffering writes to secondary logs is again to bundle them into at least one 4 KB page (similar to the primary log).

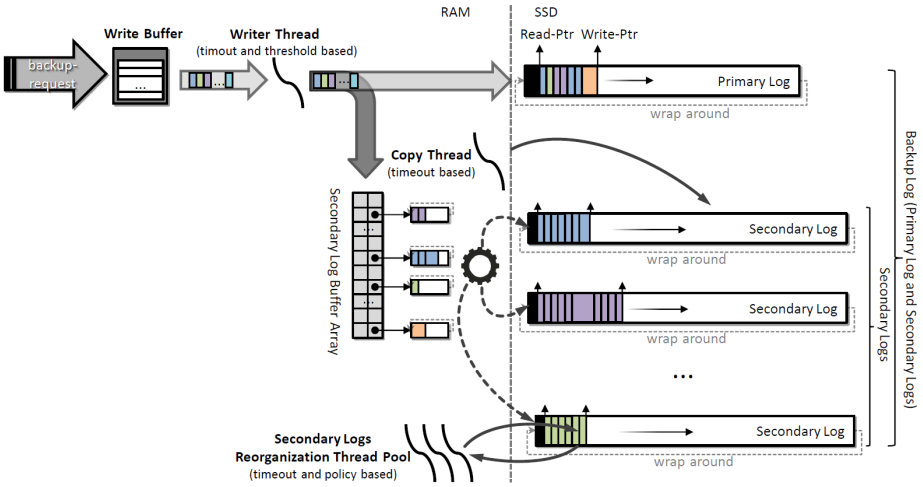

Figure 3: Log overview

It is important to note that the secondary log buffer does not introduce any risk of losing data because all backups are stored in the primary log on SSD. However, the write buffers are flushed periodically (less frequently than the primary write buffer) by the copy thread. The benefits are threefold: First, the primary log can remove all objects that are written to the corresponding secondary logs by adjusting the read pointer only. Second, the reorganization of secondary logs is more efficient with all objects in log (more in Section 5). And third, the recovery process is faster if no or only a few objects in the primary log have to be processed (more in Section 4).

In case of frequent updates from a node, backups may skip the primary log and secondary log buffers and are directly copied from the primary write buffer into the secondary log.

The secondary logs also allows us to reduce the object header per entry. First, there is obviously no need to store the NID_O in the headers because every object in a secondary log has the same NID_O. Furthermore for consecutive object IDs it is sufficient to only store the full LID of the first object and a flag for successor entries. We expect this case to occur quite often as we allocate objects with consecutive IDs on each node and expect updates to be rather seldom. Furthermore, during recovery we have to read in the full secondary log and can reconstruct consecutive IDs easily.

4 Recovery

In large-scale clusters node failures are a rule, not an exception. Therefore, DXRAM keeps the states of nodes in a log spread over backup nodes, see Section 3. Below we describe how the recovery works for a failed node.

Node failures are detected using timeouts in case of data/backup requests by any node. In addition we use the super-peer overlay (described in Section 2), to establish a hierarchical heart beat protocol between each peer and its associated super-peer. Any peer suspecting a peer failure informs the responsible super-peer, which may wait for the next heart beat. The recovery of a failed node is controlled by its associated super-peer, see Figure 4. This will allow distributing recovery control on different super-peers if several peers fail.

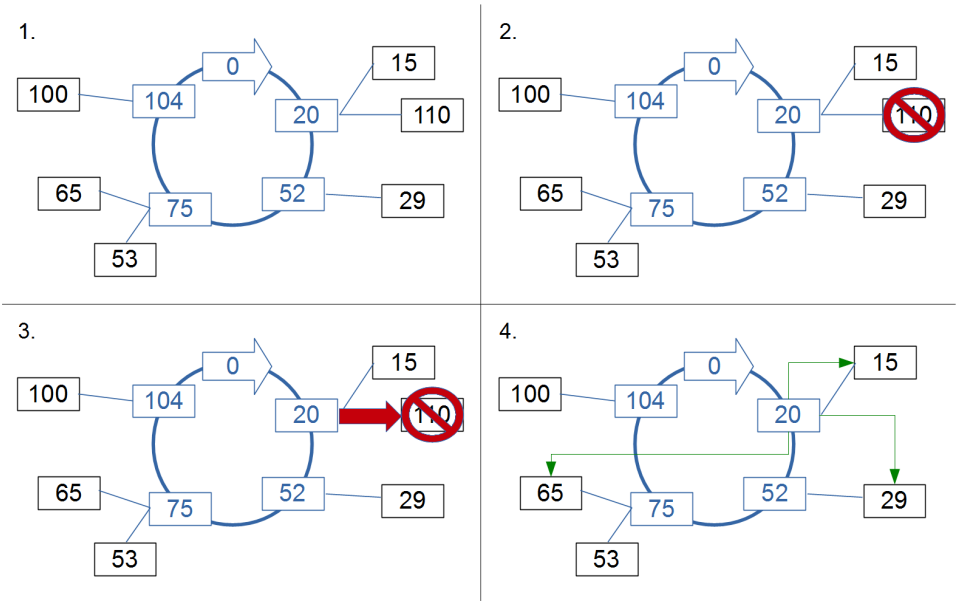

Figure 4: Recovery process: 1. Super-peer overlay with four super-peers (20, 52, 75, 104) and six peers (15, 110, 29, 53, 65, 100) 2. Failure of peer 110 3. Failure detection 4. Distribution of recovery messages

The first step of the super-peer during recovery is to determine backup nodes of the failed peer. As mentioned in Section 2, super-peers provide fast node lookup using CID ranges. These CID range entries do also include a list of backup nodes, typically 2-3 backup nodes. Thus the super-peer can easily determine, which backup nodes to contact to recover the full state of a crashed node. If objects have been migrated they belong to the node state of the new node and no longer to initial creator. As object migrations are also stored in the super-peers (resulting in a CID range split) this does not cause any problems during recovery.

If a super-peer crashes it is recovered by the super-peer overlay by a neighbor super-

peer, which is holding replicas of the meta-data. In order to not burden the system much by meta-data replication on super-peers, we only have 1-3 replica for each super-peer state. As we use dedicated nodes as super-peers, we expect enough memory for such a configuration. In rare cases several super-peers may crash simultaneously, resulting in meta-data loss. Still this disaster scenario will not lead to data loss and after the super-peer overlay structure has been repaired, the new super-peers will recover their meta-data through a multicast to gather CID ranges from their associated peers, which will also include backup nodes and migrated chunks. The chunk IDs of the latter cannot be re-used locally and thus we can use the entry in the paging-like map tables for storing the node where the chunk has been migrated to. This will however also require that if a chunk is migrated from the creator to another node and from there again to a new node, that the creator will be kept informed. This approach is also used after a full power outage, which would require to rebuilt all super-peers, which can be done in parallel like described above.

If a super-peer has gathered all backup nodes to be contacted it will always start with backup 1 and if this one is unavailable go to backup 2 and so on. The idea behind this sequential backup node selection is supported by the strict backup update order during fault-free execution, which is performed asynchronously for performance reasons. The latter ensures that the newest versions are always found in backup 1 and if unavailable we got for 2, where we still hope to find the most recent versions. Depending on the failure situation we might lose the newest version of an object because the object holder crashed and backup 1 crashed too and the other backups did not get updated because of message loss on the network. Obviously, this is a rather seldom event in a data center, it still could occur. In that situation we plan to recover some older version and inform the application. Again we want to point out that the core API will include a sync command, which allows applications to force synchronous backups for critical operations. But of course the overall system performance would be burdened if an application programmer will place after each put a sync.

After a backup node receives a recovery message for a given NID, it first flushes its write buffer and the corresponding secondary write buffer. This will ensure that all backup copies of the failed node are in its secondary log on SSD. It now depends on the configuration regarding the number of backup nodes, which will affect the size of the secondary log, e.g. if we have storage nodes with 32 GB RAM and distribute their state on 100 backup nodes, each backup node will potentially store 320 MB of data in its secondary log. To avoid pressure on the re-organization (described in the following section), we use at least double size for the secondary log, for the example above resulting in 640 MB for the secondary log per node. This is no problem as storage capacity on SSDs is increasing and getting cheaper, so we can expect at least 512 GB, which would allow each node to store logs for around 800 storage nodes. Depending on the free memory of the backup node, it might be possible to read in the full secondary log at once and analyze it in memory with multiple threads benefiting of multiple cores. Analyzing is necessary to determine the newest versions and also deleted chunks. Otherwise, if the backup node does not have enough memory to load the full secondary log, it needs to analyze it step by step, e.g. 16 MB per step.

Assuming all backup nodes have enough memory, they all could recover the secondary

log in their memories, could inform the super-peer, and could from then on serve chunk requests. This is the fastest possible recovery as this approach does run in parallel without any data transfers over the network. However, data locality is not preserved, as the state of the recovered node is now spread over numerous or many nodes. Therefore, it is planned to asynchronously move data from backup nodes to a fresh node to rebuilt the crashed node. In contrast to existing solutions, we plan to do this asynchronously, being able to serve requests, while we move larger chunk sets to the fresh node. So far we have implemented the local recovery, loading the full secondary log into the memory of the backup node and run a multi-threaded analysis.

5 Reorganization

The write buffers and the primary log are flushed periodically or when running full (beyond a given threshold), so there is no need for reorganizing them. However, the secondary logs are contiguously filled with update and delete log entries appended to the end of the log. For each chunk creation, update, and deletion we create one log entry.
Obviously, there is a strong need for reorganization in order to free space of outdated and deleted log entries, at latest when the log size reaches a predefined threshold. This is a known issue in log-structured file systems, which also benefit of the sequential high-speed write operations but have to spend considerable efforts for cleaning the log when running out of space.

Our current prototype uses a rather simple reorganization scheme. If the log size exceeds 75% of the overall capacity we trigger a log reorganization. The latter reads in a full secondary log and runs a multi-threaded cleaning using a thread pool with a work stealing mechanism. The cleaner threads analyze the full log for valid, deleted, and outdated entries. Because of the strict sequential write order we know that newer version are stored in the log after older ones.

While this basic in-memory solution performs well, we are currently studying incremental and more space-efficient approaches avoiding to load the full secondary log into memory. This in turn requires to reason more about the log layout as we cannot afford millions of small random gets and puts on SSD like in memory. Another aspect is related to the update patterns, depending on the applications. If an application updates a few objects many times with the described on-demand reorganization, the log is increasingly populated with many outdated log entries. This slows down the recovery process as all log entry headers need to be analyzed during recovery. We plan to address these problems by an incremental and periodical reorganization scheme.
In [RO92] the authors have shown that the reorganization overhead can be reduced by dividing the log into segments and monitoring write frequency of each segment. The authors then propose to prefer reorganizing segments that are updated less frequently as these segments tend to stay longer organized. This in turn avoids repeated moves of unchanged log entries and leads to a categorization of log entries depending on their update frequency (cold and hot segments). The selection of segments is based on the cost-benefit policy (equation 1) in which ”u” the utilization of the segment and ”age” the age of the youngest

log entry in that segment is. This means the desired segment contains a few, for a long unchanged entries, hence much space can be regained and the resulting new less-filled segment includes data that was changed less frequently in the past.

$$\frac{\text{benefit}}{\text{cost}} = \frac{\text{free space generated} * \text{age of data}}{\text{cost}} = \frac{(1-u) * \text{age}}{1+u} \tag{1}$$

While this strategy enables the reorganization of logs with low additional write costs there are also some shortcomings. First of all this approach requires maintaining some monitoring data in memory per log entry to determine the segment with best cost-benefit coefficient and to decide which log entries are alive. Furthermore, by reorganizing only a subset of log entries at a given time, the system can no longer determine the newest version of an log entry by its position in log (newer versions are nearer to the end of the log). Thus we need to add a version number to the log entry headers. Considering the sheer amount of chunks we expect, this might be an issue. On the other hand by giving up the strict ordering of log entries, techniques like threading and hole-plugging [MRC+97] or slack space recycling [OKC+10] are applicable.

Another interesting approach is to distribute log entries into different logs based on write statistics. This means chunks that are changed frequently are bundled in one log and those that are changed seldom in another one (more than two partitions conceivable). This creates hot and cold zones like with the cost-benefit policy but with shifting the complexity from reorganization to storing of backups. The advantage of this approach is the rather simple reorganization, because the segment selection is less important. However both variants are adaptable which will be an important aspect in finding a space and time efficient reorganization scheme for logging billions of small chunks.

6 Related Work

Logging has been used in different research areas but because of the limited space in this paper we discuss only the most relevant related work.

As DXRAM is inspired by RAMCloud [OAE+10] it is sharing many ideas but there are also important differences, including the logging approach. First of all, RAMCloud is providing a table-based data model and is not aiming at supporting billions of small data objects like DXRAM does. The in-memory layout is a 1:1 copy of the remote log which fits well for a table-based model but not as good for many small objects. The reorganization of the disk logs is controlled by the storage node reorganizing segments in memory and writing the compacted results to new segments on potentially new backup nodes. While the reorganization can run in memory this introduces a heavy network traffic. Therefore, we decided to allow backup nodes to do a reorganization for their secondary logs independent of other backup logs and the storage node. Furthermore RAMCloud does not have the two stage logging approach with the sorting of log entries per node as we have. Finally, it is worth to note that RAMCloud has successfully shown that it is possible to recover the state of node with 35 GB of in-memory data in around 1.6 seconds [ORS+11]. This is a lower

bound as the authors have used 1,000 backup nodes for recovering one node in parallel and used an Infiniband high-speed network.

The idea behind log-structured file systems like SpriteFS [RO92] is to minimize the movements of the actuator arm in traditional HDDs, because moving the actuator arm to the right position and waiting for the rotation until the desired block is accessible can take numerous milliseconds. As a consequence SpriteFS tried to write sequentially whenever possible, which in turn means that block updates are appended with higher version instead of replacing blocks in place. Read accesses are mostly served from the block cache avoiding heavy random reads on the HDD. With time the log becomes overfilled with old versions and lacks free space to store new objects or updates. SpriteFS addressed this by dividing the log into segments and execute a cleaner on segments periodically. During the cleaning, old versions are deleted and the segment is defragmented. After defragmentation the space of the old segment is freed and the remaining objects are appended to the log. In addition the cleaner does not choose segments with the same probability but uses a cost benefit policy to maximize the amount of space that is regained with every processed segment.

SFS is a more recent work transferring the advantages of a log-structured file system from HDDs to SSDs [MKC+12]. For better performance the segment size fits the block size of the SSDs. Another interesting aspect is that the hot and cold zones are generated proactively with statistics on block level to increase the efficiency of the defragmentation. It is future work for DXRAM to refine the cost benefit policy and to optimize it for billions of small data objects stored on SSDs.

Logging has also a long tradition in database systems, where typically a write-ahead log (WAL) is used. However, these logs are totally different from the one proposed in this paper. WAL logs do not store the data per se but the transaction's meta-data to apply transactions on backup databases or to repeat failed transactions. The data objects are stored on persistent memory anyway. If the log is overfilled old log entries are flushed to the database and will be overwritten. So, there is no need for a log cleaner.

Finally, there have been proposed many SSD-based key value stores, which however typically use SSDs as caches for HDDs in order to speed up data accesses. One example is Flashstore [DSL10], which uses a cuckoo-based hash-table with compact key signatures as index structure in RAM to improve access times for reading objects from SSD. While the related insights are interesting for meta-data management in DXRAM, they do not affect the log architecture.

7 Conclusion

DXRAM is a distributed in-memory system that is designed to manage billions of small data objects. By keeping all data always in memory it is providing low-latency data access to all data, while at the same time relieving programmers from keeping caches and secondary storage synchronized. The latter requires an efficient asynchronous logging of in-memory data on remote flash disks for providing fast recovery from node failures and

avoiding data loss.

We have presented a novel two stage logging approach sorting log entries per node allowing fast recovery of failed nodes. The latter is also supported by spreading the state of one node on potentially many backup node logs, which allows a parallel recovery from many nodes. As known from the systems literature, sequential logging is fast but requires a cleaner to remove outdated and deleted data and compact the log to free space. We believe that the cost-benefit approach like proposed for log-structured files systems is a good base for a more optimized custom solution for an in-memory based storage.

Future work includes the design of cleaner policies as well as evaluations with different application access patterns (including recovery times depending on different configurations). Currently, we are adopting the BGbenchmark [BG13], which allows to run actions of social network applications on top of different storage technologies (e.g. MongoDB and SQL-X with memcached). We plan to use this benchmark for comparing DXRAM performance with key-value and distributed in-memory caches (e.g. Gemfire and Hazelcast) as well as the recovery times for single and multiple node failures.

References

[AXF+12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[BAC+13] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 49–60, Berkeley, CA, USA, 2013. USENIX Association.

[BG13] Sumita Barahmand and Shahram Ghandeharizadeh. BG: A Benchmark to Evaluate Interactive Social Networking Actions. In *CIDR*, 2013.

[CKZ09] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 181–192, New York, NY, USA, 2009. ACM.

[DSL10] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: high throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1-2):1414–1425, September 2010.

[KS13] F. Klein and M. Schoettner. Dxram: A persistent in-memory storage for billions of small objects. In *Proceedings of the 14th International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT 13, 2013.

[MKC+12] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the*

10th USENIX Conference on File and Storage Technologies, FAST'12, pages 12–12, Berkeley, CA, USA, 2012. USENIX Association.

[MRC+97] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the Performance of Log-structured File Systems with Adaptive Methods. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 238–251, New York, NY, USA, 1997. ACM.

[NFG+13] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[OAE+10] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, January 2010.

[OKC+10] Yongseok Oh, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Optimizations of LFS with Slack Space Recycling and Lazy Indirect Block Update. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, pages 2:1–2:9, New York, NY, USA, 2010. ACM.

[ORS+11] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[RO92] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.

[SWL13] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, NY, USA, 2013. ACM.

[SWW+12] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient Subgraph Matching on Billion Node Graphs. *Proc. VLDB Endow.*, 5(9):788–799, May 2012.