

Object Storage on Berkeley DB¹

Casey Marshall
University of California, Santa Cruz
CMPS 229: Storage Systems

Abstract

We have implemented an object storage system, intended to simulate the operation of an Object Storage Device (OSD), on Berkeley DB, a simple database library distributed by Oracle Corporation (formerly Sleepycat Software) [1]. This object storage layer is used in the Ceph [2] distributed file system as an alternative to the EBOFS object storage system. Our results indicate that object storage systems built on Berkeley DB provide good performance for many small objects, and competitive performance for large objects; integrating this system into Ceph has revealed a few performance problems.

1 Introduction

Ceph is an advanced, distributed, scalable file system being developed at UCSC. Ceph uses smart object storage devices managed by an object distribution and lookup layer called RADOS (reliable, autonomous, distributed object store). The current object store implementation in Ceph is an extent and B-tree based block management system called EBOFS [2].

Object storage systems like EBOFS, which provide fairly simple key/value mappings, turn out to be very similar to the interfaces provided by database systems like Berkeley DB. Berkeley DB [1] also happens to provide a number of benefits over a custom solution, since it is a very mature product, it has robust transaction and recovery support built-in, and writing an object store implementation itself is simple.

Our project was to implement an object storage layer on top of Berkeley DB, and to see how it compares to EBOFS in performance. Given that it

took one person less than three months to write a fairly stable and efficient object store, the benefits of using an existing, stable system are quite clear. We also believe that robust, simple, and scalable storage systems built on top of database systems such as Berkeley DB are feasible.

2 Implementation

The implementation of our object store is as an implementation of the C++ class `ObjectStore`, that Ceph uses internally for its object storage. The `ObjectStore` interface provides a simple set of commands that emulate the command set of the T10 object storage specification [3]. These commands fall into the following logical groups:

1. Object storage commands. These include basic operations for reading, writing, and querying objects. Objects are plain byte sequences, and are referenced by 128-bit object identifiers.
2. Collection commands. Collections are bags of object identifiers, and are referenced by 32-bit collection identifiers. Object identifiers may be added or removed from a collection, and the contents of a collection can be queried.
3. Attributes. These are simply key/value pairs, with simple arbitrary-length keys and byte sequences for values. Both objects and collections may have attributes assigned to them, so an attribute key is an object identifier or collection identifier paired with a simple key.

We call our implementation “OSBDB.” OSBDB was, relatively speaking, simple to implement, and ignoring the size and complexity of Berkeley

¹ This work was done in partial fulfillment of the requirements for the UCSC course CMPS 229: Storage Systems, Winter quarter 2007, Dr. Carlos Maltzahn.

DB itself, it has much less code to maintain and debug, a fact illustrated in table 1:

Store	SLOC
ebofs	6,600
osbdb	2,161

Table 1: Lines of C++ code in the ebofs and osbdb object stores (as measured by SLOCCount [4]).

2.1 Objects

Berkeley DB has a simple application interface, providing obvious methods such as `get`, which takes a key argument and returns the value mapped to that key, and `put`, which takes key and value arguments and creates a mapping between the two. Both keys and values are byte sequences of arbitrary length up to 4GiB.

Because this interface closely matches the object storage interface, objects are keyed directly by the 128-bit object identifier. In Ceph, object identifiers have some amount of structure, but at the `ObjectStore` level we ignore this, and pass the 128-bit value (that is, the raw C++ struct) as the key. Object values are just as simple, and are stored as-is; however, Berkeley DB offers no simple way to query the size of a mapped object without reading it, so in our implementation we include an additional “inode” record for each object, which has the form:

```
struct stored_object {
    uint32_t length;
}
```

These “inode” records are mapped by a 17-byte key formed by taking the 16-byte object identifier, and appending a single byte ‘i’.

2.2 Collections

Collections are, in principle, simple bags of object identifiers. In OSBDB, a collection is stored as a sorted array of object identifiers, represented by the structure:

```
struct stored_coll {
    uint32_t count;
    object_t *objects;
}
```

Where `object_t` is the 128-bit object identifier type.

Insertion and deletion of members is done by performing a binary search for the insertion point or object identifier to be deleted, then performing a `memmove` up or down by 16 bytes. The `ObjectStore` interface also requires that all collections be enumerated, so in OSBDB we keep a master list of all valid collection identifiers. This list has a form similar to a collection:

```
struct stored_colls {
    uint32_t count;
    coll_t *colls;
}
```

Where `coll_t` is a 32-bit collection identifier. The storage of collection identifiers is equivalent to a collection: we store them as a sorted list. This master collection list is referenced by the one-byte key ‘c’.

2.3 Attributes

Attributes are rather unfriendly, since they both add an extra overhead to the database management (including keeping a list of attributes, so they can be removed when the object is deleted), and also present a problem when dealing with variable-length keys. Our solution is to keep a list of attribute keys for each object and collection, keyed by the object or collection identifier appended with the byte ‘a’ (making for a 17-byte or 5-byte identifier). Both attribute lists have the form:

```
struct attr_list {
    uint32_t count;
    attr_key *key;
}
```

The `attr_key` type is simply a wrapper structure around a 128-byte array. This means that attribute keys are currently limited to 128 bytes, and that keys are stored padded with as many NUL bytes

to fill the buffer. Since the `ObjectStore` code only uses NUL-terminated strings for attribute keys, this solution works out well. The attribute keys are sorted lexicographically, and insertion/deletion works similarly to the collection lists.

Attributes themselves are keyed by appending the 128-byte attribute key to the object identifier or collection identifier, for referencing an object attribute or collection attribute, respectively. Attribute values are stored directly.

2.4 Summary

To summarize briefly, keys in OSBDB fall into the following categories, which have unique lengths, or at least have unique suffixes:

- Object identifiers, 16 bytes.
- Object inode identifiers, 17 bytes (final byte is always ‘i’).
- Collection identifiers, four bytes.
- Master collection list identifier, one byte (‘c’).
- Object attribute list identifiers, 17 bytes (final byte always ‘a’).
- Collection attribute list identifiers, five bytes (final byte always ‘a’).
- Object attribute keys, 272 bytes.
- Collection attribute keys, 260 bytes.

Using this partitioning of the key space, we can easily make the object name spaces independent, largely by simply using the length of keys.

3 Object Store Performance

For an initial test, we wrote a micro-benchmark program around the `ObjectStore` interface that repeatedly writes out a number of uniformly-sized objects, remounts the file system, then reads these objects in again, in a different order. We ran this benchmark with a variety of object sizes and object counts.

The read and write large object throughput of EBOFS, OSBDB, and OSBDB configured with the Btree database type, given different object

sizes, is shown in figure 1. Each test was run 1,024 times, writing out two objects of the same size, and the mean bandwidth was computed over all runs.

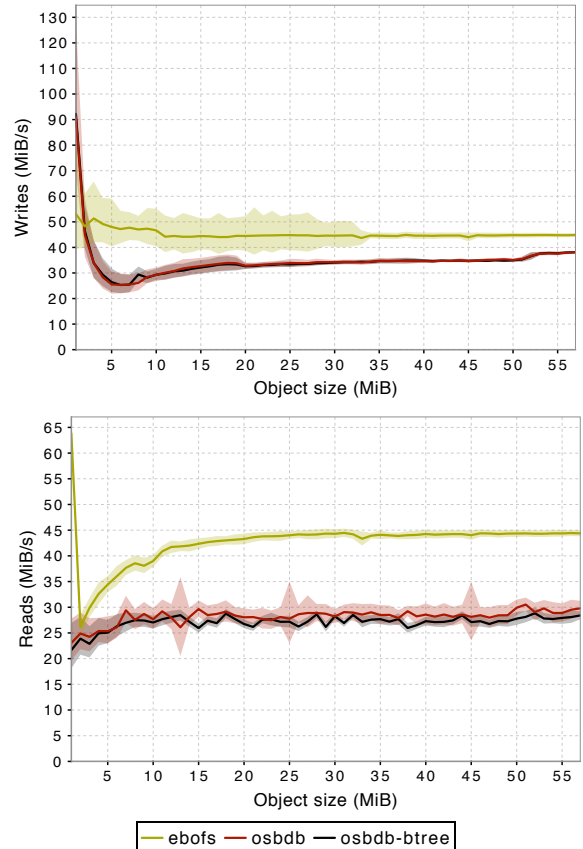


Figure 1: write (top) and read (bottom) throughput for a few large objects. The lighter band indicates the standard deviation away from the mean, plotted by the solid line.

This benchmark, like all others in this paper, was run on an Ubuntu Linux 6.10 system running on an Intel Pentium D 2.8GHz processor, 1GiB of memory, and an internal 80GiB, 7200 RPM disk on the SATA bus. EBOFS used a raw disk partition, and OSBDB used a dedicated ext2 file system that was unmounted and remounted between operations.

We see generally stable throughput for all the stores, with EBOFS outpacing OSBDB for nearly all tests. The hash database type produces some interesting spikes in the variance, at seemingly logarithmic intervals.

We also ran a similar test with increasing object count, with a fixed object size of 1,024 bytes. This test was run 128 times for each object count, and the mean 1,024 object throughput is presented in figure 2. The decreasing throughput of the Berkeley DB hash database type may be caused by re-hashing the database hash table as the object count grows; BDB offers control over the hash size and fill factor, which we haven't experimented with yet. Berkeley DB overall does very well in this test:

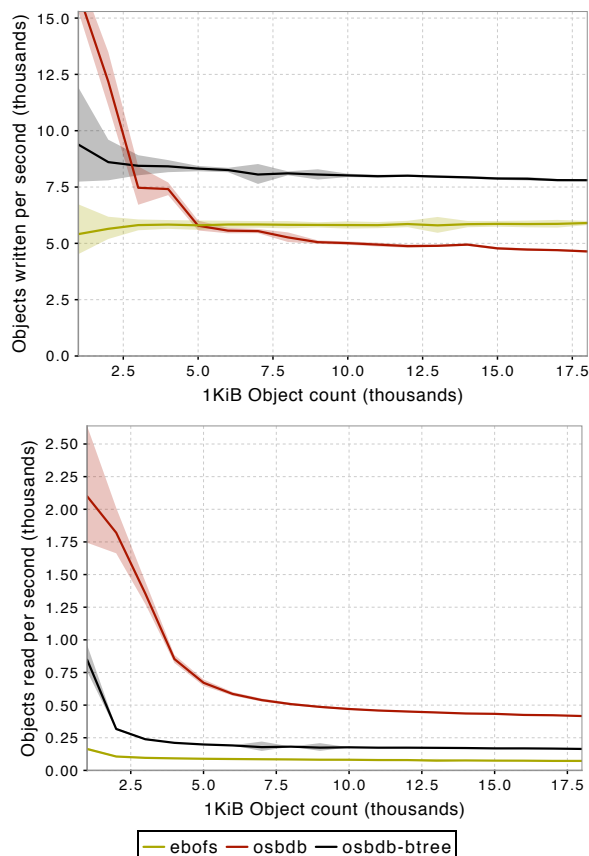


Figure 2: write (top) and read (bottom) throughput, in thousands of objects per second, given object stores of increasing utilization. The lighter band again denotes the standard deviation away from the mean, plotted by the solid line.

One conclusion to draw from these benchmarks is that EBOFS performs slightly better for very large

objects (although, this is not so great a concern in Ceph) and that the Berkeley DB-based object stores scale rather well for stores that contain many small objects. Since most objects in Ceph are around 1MiB or less, OSBDB should be well suited for it.

4 Ceph Benchmarks

For these benchmarks we used the *fakesyn* test program included in Ceph. Fakesyn supports a number of synthetic workloads, which run in a single process. For these benchmarks, we instrumented the *SyntheticClient* class to record how long the operation performed (omitting start-up and shut-down times) took to run. These benchmarks also simply used the default cache sizes for each store, and made no attempt to flush system or in-program caches during the run. Here we present comparisons for runs of some of these workloads.

writefile and **readfile** are workloads that write out a file in chunks, then read that file back in again. In figure 3, we show a run of **writefile** followed by **readfile** for a 1GiB file, reading and writing 256KiB and 1MiB chunks. EBOFS has a clear advantage on reads, while both kinds of Berkeley DB store achieve good write performance²:

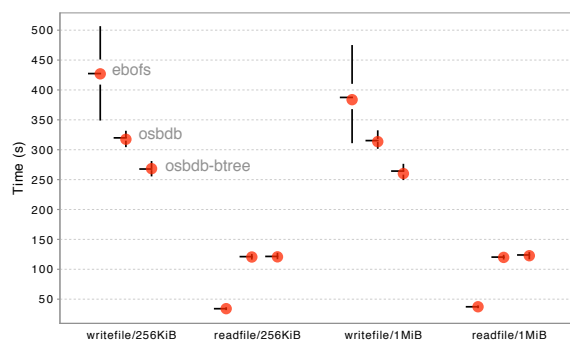


Figure 3: readfile and writefile, operating on a 1GiB file in 256KiB and 1MiB chunks. These tests represent 64 runs of the workload.

This seems to contradict the results in the previous section, where we found that OSBDB does very well for small objects.

² The charts in this section are all quartile plots; the dot denotes the median, the horizontal line the mean, and the verticals the region from the minimum to the first quartile, and the third quartile to the maximum.

The situation, in fact, degrades considerably for OSBDB if the block size is reduced to a few kibibytes:



Figure 4: readfile and writefile, operating on a 1GiB file in 1KiB and 4KiB chunks. The above represents sixteen runs of the workload.

makedirs is a workload that creates a directory tree with files scattered about, given a directory count, number of files, and directory depth. **walk** walks through the entire created hierarchy; **readdirs** reads the directory hierarchy (**walk** and **readdirs** are, therefore, very similar tests). The **makedirs** results — with the count, file count, and depth 2, 16, and 4, respectively (which means a binary file tree of depth 4 with 16 files in every directory) — speak unfavorably for OSBDB:

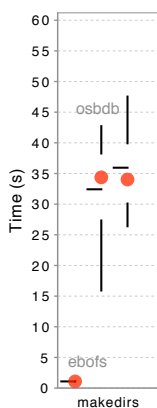


Figure 5: results of running makedirs with a count of 2, a file count of 16, and a depth of 4. Averaged over sixteen runs.

Reading these directories again, however, shows a better picture:

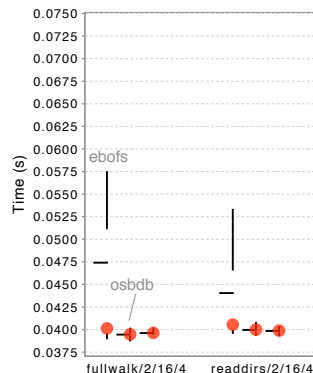


Figure 6: results of running walk and readdirs on the hierarchy created with makedirs. Averaged over sixteen runs.

We are not certain what the cause of these performance differences is. One possible cause is the relative inefficiency of *partial* reads and writes in Berkeley DB; also a problem may be our implementation of collections, and the rather large number of data copies that must be made to perform certain operations. Locking in OSBDB is also rather coarse-grained in this implementation, even though Berkeley DB supports shared and exclusive per-record locks.

5 Related Work

Ceph already has a fairly robust and efficient object storage system called EBOFS [2]. Possibly the chief advantage OSBDB has over EBOFS is that it is simpler and easier to maintain, and provides excellent support for transactions, logging, concurrency, recovery, and “hot” backups.

Many systems have used Berkeley DB, and its predecessors from the original dbm systems on Berkeley UNIX, such that it is impossible to enumerate them all here. Notable free software packages that use Berkeley DB for storage include the Cyrus mail server [5], the Subversion version control system [6], the MySQL database (up until version 5.1) [7], and the OpenLDAP directory server [8]. Many organizations have employed Berkeley DB for back-end storage in private systems.

The PVFS2 system, the underlying storage system for the pNFS extension to the NFSv4 protocol, uses Berkeley DB for file metadata storage [9]. The Panasas ActiveScale cluster management system and the Veritas Enterprise Administrator object bus are two enterprise storage systems that use Berkeley DB [10].

6 Future Work

We consider this project a success; we have shown that it is possible to write an efficient storage system very quickly using Berkeley DB. We have, however, only scratched the surface of what is possible in this area.

Examples of things to consider include the transaction and logging support in Berkeley DB, and investigating good ways to incorporate these features into a storage system. Backups and hot copies, which Berkeley DB supports, offer a useful way to manage backups and replication in a storage system.

While we have developed OSBDB in the context of Ceph, storage systems built specifically around Berkeley DB is an interesting realm of research, and such systems are useful in commercial and enterprise storage systems.

7 Conclusions

We have implemented an object storage interface using the embedded database Berkeley DB. We have found that even though the code we had to write was simple and straightforward, our object storage system has good performance characteristics, even when compared against an advanced, complex system dedicated to this purpose. There are some clear performance issues remaining, but we believe that there is a simple cause underlying these problems. Robust, simple systems such as Berkeley DB, which provide fast and safe object storage, are compelling bases for object-based storage systems.

OSBDB is available as a part of Ceph, which is free software. Ceph and OSBDB can be downloaded from, and you can join the project at, <http://ceph.sourceforge.net/>.

References

- [1] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the 1999 Annual USENIX Technical Conference*.
- [2] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*.
- [3] R. O. Weber, editor. SNIA Project Specification T10/1355-D: Information Technology; SCSI Object Based Storage Device Commands (OSD).
- [4] D. Wheeler. SLOccount. <http://www.dwheeler.com/sloccount/>.
- [5] Carnegie Mellon University. Project Cyrus. <http://cyrusimap.web.cmu.edu/>.
- [6] CollabNet. Subversion. <http://subversion.tigris.org/>
- [7] MySQL AB. MySQL Database System. <http://mysql.com/>.
- [8] OpenLDAP Foundation. OpenLDAP. <http://www.openldap.org/>.
- [9] D. Heldebrand and P. Honeyman. Exporting Storage Systems in a Scalable Manner with pNFS. In *Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technology (MSST 2005)*.
- [10] Oracle Corporation. Building Robust Storage Systems with Oracle Berkeley DB. White paper. October 2006.
- [11] D. Gilbert et al. JFreeChart. <http://www.jfree.org/jfreechart/>.