Benchmarking and Testing OSD for Correctness and Compliance

Dalit Naor *Petra Reshef* *Ohad Rodeh* *Allon Shafrir* *Adam Wolman*
Eitan Yaffe
IBM Haifa Research Lab
{*dalit, petra, orodeh, shafrir, wolman, eitany*}*@il.ibm.com*

Abstract

Developers often describe testing as being tedious and boring. Our work challenges this notion. We describe tools and methodologies crafted to test object-based storage devices (OSDs) for correctness and compliance with the T10 OSD standard. Special consideration is given to testing the security model of an OSD implementation. Some work was also carried out on building OSD benchmarks. This work can serve as a basis for a general-purpose benchmark suite for OSDs in the future, as more OSD implementations emerge.

The tool described here has been used to verify object-disks built by Seagate and IBM Research.

1 Introduction

Developers often find testing tedious and boring. Our work attempts to challenge this notion. We describe a tool set created to test object-storage-devices (OSDs) for correctness and compliance with the T10 OSD standard [14, 11], which proved to be a difficult and challenging part of our overall development effort. Aside from correctness, performance is an important quality of all implementations; initial work was carried out on benchmarking OSDs.

Generally speaking, the T10 standard specifies an object-disk that exports a two-level object-system through a SCSI based protocol. The OSD contains a set of partitions, each containing a set of objects. The objects and partitions support a set of attributes. Protection is provided by a credential-based security architecture with symmetric keys.

Our group at IBM Research built an OSD together with a test framework; a description of our object-store related activities can be found in [5]. For the test framework, we faced several choices ranging from white-box to black-box testing. Black-box testing was selected as the primary methodology because it would keep the testing infrastructure independent of the OSD implementation. However, in addition to the black box testing, we developed some limited capabilities based on gray-box techniques to test and debug our own OSD. Building upon knowledge of the internals of the target implementation, this had the potential to considerably improve coverage.

Our goal was to build a small, light, tool set that would achieve a good coverage. A `tester` program was written to accept scripts containing OSD commands. The `tester` sends the commands to the target OSD and then checks the replies, thereby creating a certain workload on the target.

This kind of testing falls under the sampling category. From the possible scenarios of commands that reach the target, only a sample are tested. Passing the tests provides a limited guarantee of compliance and correctness; however, it does not prove there are no bugs lurking in the code. The idea is to identify and test the subset of scenarios that will provide the best possible coverage. To address the sampling issue and increase coverage, we wrote a `generator` program to generate scripts with special characteristics.

Black-box testing proved a fortunate choice later on. During 2005, we were part of a larger IBM Research group that built an experimental object-based

file system. This system was demonstrated at Storage Networking World in the spring of 2005 [4]. The file system was specified to work with any compliant OSD. To demonstrate this, we worked with Seagate^{TM} who provided their own OSDs. Our group was commissioned to test the correctness and compliance of our own target as well as Seagate's. This was a necessary prerequisite before wider testing within the file system could be carried out.

The main contribution of this paper is to report on our practices and experience in testing object stores that are compliant with the new OSD T10 standard. As an emerging storage technology, object storage is still in its infancy, but is expected to gain momentum in the near future. So far, very few OSD implementations have been reported, and even fewer are compliant with the OSD standard. Hence, we believe that our work regarding the validation of such implementation and conformance with the standard will be relevant and valuable to the community at large. In this paper, we also argue and demonstrate that although close in spirit to a file system, T10 compliant OSDs have unique characteristics that distinguish their testing and validation from that of traditional file systems. One of the most notable differences is the OSD security model and its validation.

This paper describes the tools developed, and provides specific examples that emphasize how these tools were tailored to address the specific difficulties in testing an OSD, both for compliance and correctness. The rest of the paper is organized as follows. Section 2 describes the T10 specification and walks through some of the difficulties it poses for testing. Section 3 describes the testing infrastructure. Section 4 describes the techniques used to locate bugs. An important aspect of the OSD T10 protocol that requires special testing tools and techniques is the OSD security model. Section 5 describes the mechanism developed to test the security aspects of OSDs. Section 6 talks about the benchmarks devised to measure performance. Section 7 describes related work and Section 8 summarizes our findings.

2 The OSD specification

2.1 T10 overview

An object-disk contains a set of partitions, each containing a set of objects. Objects and partitions are identified by unsigned 64-bit integers. Objects can be created, deleted, written into, read from, and truncated. An additional operation that was needed for the experimental object-based file system demonstrated in [4] and may be standardized in the near future is `clear`. Clearing means erasing an area in an object from a start offset to an end offset. Partitions can be created and deleted. The list of partitions can be read off the OSD with a list operation. A list operation is much like a `readdir` in a file system, where a cursor traverses the set of partition-ids in the OSD. Similarly, the set of objects in a partition can be read by performing a list on the partition.

Partitions and objects have attributes that can be read and written. A single OSD command can carry a list of attributes to read and a list of attributes to write, among other things. There are compulsory attributes and user-defined attributes. Compulsory attributes are, for example, object size and length. User-defined attributes are defined by the user outside the standard. There are no size limitations on such attributes. For brevity considerations, user-defined attributes are not addressed here.

A special root object maintains attributes that pertain to the entire OSD. For example, the used-capacity attribute of the root counts how much space is currently allocated on the disk.

An important aspect of a T10 compliant OSD is its security enforcement capabilities. The T10 standard defines a capability-based security protocol, based on shared symmetric-keys. The protocol allows a compliant OSD to enforce access-control on a per-object basis, according to a specified security policy. This enforcement is done using cryptographic methods. The protocol allows capability revocations and key refresh. The standard protocol defines three different methods for performing the validation, depending on the underlying infrastructure for securing the network. Of these, we only consider the CAPKEY method.

2.2 Difficulties

The T10 specification poses several serious difficulties for testing tools. Three examples are highlighted in this section: testing atomicity and isolation guarantees, testing parallelism, and verifying quotas.

Atomicity and isolation guarantees are weak in order to provide better performance; this creates non-determinism. For example, assume two writes W_1, W_2 are sent to the same extent in an object. The result is specified as some mix of the data from W_1 and W_2. This mix might be limited by the atomicity provided by the OSD, which is implementation-dependent.

Parallelism. Commands sent in parallel to an OSD can be executed in any order. For example, if a command C_1 = create-object(`o`) is sent concurrently with C_2 =delete-object(`o`), there are two possible scenarios.

1. The OSD performs C_1 and then C_2. The object is created and then deleted. Both commands return with success.

2. The OSD performs C_2 and then C_1. The object is deleted and then created. The delete fails because the object did not exist initially. C_1 returns with success; C_2 returns with an `object-does-not-exist` error. Object `o` remains allocated on the OSD.

In general, the non-determinism that results from concurrently executing multiple commands on the OSD poses a big challenge on its verification.

Quotas, which pose another kind of problem, are specified as being fuzzy. For example, consider an object with a certain quota limit. If the object-data exceeds the quota limit, the target *must* signal an out-of-quota condition upon the next write into the object. However, it *may* signal, at its own discretion, an out-of-quota–even if the written data is less than the quota but 'close' to it by a certain confidence margin. The upshot is that it is not possible to write a simple test to check for quota enforcement.

Another issue is that object, partition, and LUN used-capacity are not completely specified. In this discussion, we focus solely on objects. An object's used capacity is defined to reflect the amount of space the object takes up on disk, including metadata. However, an implementation has freedom in its usage of space. For example, in one implementation, one byte of live data may consume 512 bytes of space, whereas in another implementation it may consume 8192 bytes (*i.e.*, one 4K page for its metadata an one 4K page for data). Since various sizes are legal, a single one-size-fits-all test is impossible to devise.

3 Infrastructure

3.1 Components

Our OSD code, including the testing infrastructure, is structured as follows (see Figure 1):

- `tester`: a relatively simple program that reads scripts of OSD commands, sends them to the target, and verifies their return values and codes.

- iSCSI OSD initiator: an addition to the Linux kernel of T10-specific iSCSI extensions. Specifically, this refers to bidirectional commands, extended CDBs, and the T10 command formats[9].

- iSCSI target: a software iSCSI target.

- Front-End (FE): module on the target that decodes T10 commands.

- Reference implementation (simulator): the simplest OSD possible.

- Real implementation (OC): an optimized OSD.

While building the components, we followed the engineering principle of leveraging a small, well tested, module to test and verify a larger module. The simpler the module, the more confidence we had in it. We aimed to keep testing-modules simple and with low line counts.

3.2 Tester and script language

The design point was to build a simple `tester` program. We were interested in minimizing the

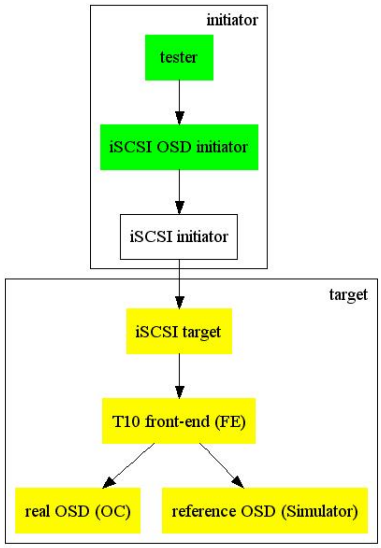

Figure 1: OSD code structure: the set of components

amount of state kept on the tester side. Minimizing the state would simplify the tester and improve its reliability[1].

A script language was tailored specifically for the OSD T10 standard. The commands fall into two categories. The first category contains simple T10 commands such as `create`, `delete`, `read`, `write`, and `list`. Each command can be accompanied by the expected return code and values. The second category contains composite commands, such as the device snapshot command. This command takes a snapshot of the entire contents of the target or of a specific object. This ability is used to compare the contents of the target with other targets, such as the reference implementation target or a different target implementation.

The commands are grouped into blocks, which can be defined recursively. The types of blocks are:

- Sequential block: The `tester` waits for each command to complete before submitting the next command in the block.
- Parallel block: Commands in the block are submitted to the target concurrently. It is the responsibility of the script writer to ensure valid scripts (*e.g.*, creating a partition must terminate successfully before creating an object within it, if the object creation is expected to complete successfully). The order in which the commands are sent is not defined. For example, the iSCSI layer may change the order in which it sends the commands.

A simple example is:

```
create oid=o1;
write oid=o1 ofs=4K len=4K;
delete oid=o1;
```

This script creates an object, writes 4K of data into it, and then deletes it.

A more complex script is:

```
par {
  create oid=o1;
  create oid=o2;
}

par {
  write oid=o1 ofs=20K len=4K;
  write oid=o2 ofs=8K len=4K;
}
```

This script creates two objects concurrently and then writes into them concurrently.

One can use the `seq` operator to create sequential blocks:

```
create oid=o2;

par {
  seq { create oid=o1; write oid=o1;
        delete oid=o1 }
  clear oid=o2 ofs=30K len=512;
}
```

The `tester` execution phases are:

- Parse script, building a DAG (directed acyclic graph) representing command dependencies.
- Submit commands according to the specified order, handling target responses for each command and verifying the result.

[1]The total line-count for the `tester` is about 8000 lines of C code

3.3 Reference implementation

We built a reference-implementation, or *simulator*. The simulator was the simplest OSD implementation we could write. It uses the file system to store data, where an object is implemented as a file and a partition is implemented as a directory. The simulator core is implemented with 10000 lines of C code. Incoming commands are executed sequentially; no concurrency is supported.

3.4 Script generator

A *script-generator* automatically creates scripts that are fed into the `tester`. The generator accepts parameters such as error percentage, create-object percentage, delete-object percentage, etc. It attempts to create problematic scenarios such as multiple commands occurring concurrently on the same object. The expected return code for each command is computed and added to the script. When the script is executed, the tester verifies the correctness of the return codes received from the target. Generated scripts are intended to be deterministic so they produce verifiable results. However, non-deterministic scripts are also useful for testing the stability of the target to make sure it doesnt crash.

3.5 Gray-box testing

For gray-box testing, we added *crash command*, *configurations*, and *harness-mode*. These are specific to our implementation and are not generic for all OSDs. We modified the target OSD as little as possible to allow for gray-box testing. This would ensure that the tests were measuring something close to the real target behavior.

The crash command is a special command outside the standard command set. Used for testing recovery, it causes the OSD to fail and recover. We believe it should be added to the standard in order to enable automated crash-recovery testing.

A configuration file contains settings for internal configuration variables, such as the number of pages in the cache, the number of threads running concurrently, and the size of the s-node cache. Running the same set of tests with different configuration files yields better coverage with little expense. Specifically, assuming bugs occur at small configurations, one can focus only on the small configurations.

Harness-mode is a deterministic method of running the OSD target. The internal thread package and IO scheduling is switched to a special deterministic mode. This mode attempts to expose corner cases and race-conditions by slowing down or speeding up threads and IOs. The `tester` program is linked directly with the target, thereby removing the networking subsystem. This creates a *harness* program. The harness can read and execute a script file. The full battery of tests is run against the harness. If a bug is found, it can be reproduced deterministically.

We used a regression suite composed of many scripts in testing. The regression contains short hand-crafted scripts that test simple scenarios and large 5,000 - 10,000 line tests created with the script-generator.

4 Techniques

This section describes a number of techniques employed by the testing suite to test and verify non-trivial properties of an OSD implementation. The techniques proved to be extremely useful in identifying bugs in the system and as debugging tools. The techniques that were developed to test the security aspects of an implementation are discussed separately in Section 5.

4.1 Verifying object data

Data written to objects requires verification. We used a two pronged approach to check whether the on-disk data is equivalent to the data written into it by the user. A lightweight verification method of *self-certifying* the data was employed for all reads and writes, and a heavyweight method of *snapshots* was used occasionally.

The lightweight method consisted of writing *self-certifying data* to the disk and verifying it when the data is read back. For writes that are 256 bytes or more, the `tester` writes 256-byte aligned data into objects. At the beginning of a 256-byte chunk, a header is written containing the object-id and offset. When reading data from the OSD, the `tester` checks these headers and verifies them. Because the

data is self-certifying, the `tester` does not need to remember which object areas have been written to. A complication arises with *holes*. A hole is an area in an object that has not been written to. When a hole is read from disk, the OSD returns an array of zeros. This creates a problem for the `tester` because it cannot distinguish between cases where the area is supposed to be a hole and cases where the user wrote to the hole but the target "lost" the data.

Snapshots are the heavyweight method. A snapshot of an object disk is an operation performed by the `tester`. The `tester` reads the whole object-system tree off the OSD and records it. In order to verify that a snapshot is correct, the `tester` compares it against a snapshot taken from the simulator. Technically, the object-system tree is read by requesting the list of the partitions and then the list of the objects in each partition. All the data, including attributes, from the root, partitions, and objects, is read using read and get-attribute commands.

Using snapshots helps the `tester` avoid having to keep track of the state of the target. The `tester` just needs to compare its state against another OSD. Theoretically, if we had n different implementations, we could compare them all with each other. In practice, the regression suite runs against three different implementations: harness, simulator, and real-target. The snapshots are compared to each other.

The problem of verifying object data is very similar to verifying file data, and therefore the two abovementioned methods are similar to standard practices in testing file systems, except for the treatment of holes.

4.2 Crash recovery

Recovery is a difficult feature to verify because it exhibits an inherently non-deterministic behavior. For each command that was executing at the time of the failure, the recovered OSD state may show that: it had not been started, it was partially completed, or it finished completely.

To cope with this problem, we allow for checking the consistency of only a partial section of the system. For example, in the script

```
par {
  seq { create oid=o1;
        write oid=o1 ofs=4K len=512;
        crash }
  seq { create oid=o2;
        write oid=o2 ofs=20K len=90K
        }
  seq { create oid=o3;
        write oid=o3 ofs=8K len=8K }
}

snap_obj oid=o1;
```

three objects, `o1, o2, o3`, are created and written to. After the write to `o1` completes, the OSD is instructed to crash and recover. Finally, the snapshot from object `o1` is taken. It is later compared to the snapshot from the reference implementation. The state of objects `o2` and `o3` are unknown; they may contain all the data written to them, some of it, or none of it. In fact, they may not exist at all.

4.3 List command

The specification of the `list` command is very lax in its definitions of consistency. For example, if a list operation is performed on a partition, and objects are created concurrently, the said objects can show or not-show up in the list. This makes testing difficult.

We took a two pronged approach with list-testing. At points where the state of the OSD is deterministic, a snapshot is taken and compared against the reference implementation. This tests the list operation because taking the snapshot involves a listing of the partitions. Additionally, some of the scripts in the regression suite send the list concurrently with other commands–without verifying the returned results.

4.4 Bug hunting

The *Script-generator* program is a very useful tool. Its primary use was to increase testing coverage via automatic generation of interesting tests. However, it also turned out to be a useful debugging tool. Hunting for a bug that was initially identified with a very long and automatically generated script (say, 10,000 lines) required the generation of many shorter scripts. Instead of manually generating the

short scripts, we could use the generator's input arguments wisely to produce scripts that narrow the search space.

5 Testing of security mechanisms

The T10 OSD standard specifies mechanisms for providing protection. We refer to these as the *security mechanisms* (corresponding to [14, Section 4.10] on *policy management* and *security*). This section discusses the techniques used to test the security mechanisms implemented by an OSD target, specifically the CAPKEY security method.

Most of the tested security mechanisms have to do with the validation of commands. Testing the implementation for correct validation is extremely important as a minor inconsistency of an OSD target with the specification may violate all protection guarantees.

5.1 Validation of an OSD command

The OSD standard uses a credential-based security architecture. Each OSD command carries an additional set of *security fields* to be used by the security mechanisms. We refer to this set of fields as a *credential*; this is a simplification, for the sake of brevity, of the credential definition as specified in the OSD standard.

Every incoming command to the OSD target requires the following flow of operations to be executed by the target:

1. Identify the secret key used to authenticate the command. This stage requires access to previously saved keys.

2. Using the secret key, authenticate the contents of all command fields (including the security fields).

3. Test that the credential content is applicable to the object being accessed. This stage requires access to previously saved attributes of the object.

4. Test that all actions performed by this command are allowed by the credential.

The T10 OSD standard specifies the exact content for each security field in the command. A target permits the execution of a command if all security fields adhere to this specification.

We say that a credential is *good* if (1) it is valid and (2) its appropriate fields permit the command that 'carries' it. We say that a credential is *bad* if it is either invalid (*e.g.*, has bad format or is expired) or if it does not permit the operations requested by the command that 'carries' it. *Valid Commands* are commands that carry good credentials, whereas commands that carry bad credentials are considered *invalid*.

5.2 Testing approach

For a given command and a given target state, there may be many possible good credentials and many possible bad credentials. A perfect testing suite should test:

For every OSD command:

For every possible target state:

1. Send the command with all possible good credentials.
2. Send the command with all possible bad credentials.

The general problem of increasing coverage, whether command coverage or target state coverage, is addressed in Section 3. There we describe how a clever combination of the `tester` together with the `script-generator` can yield increased coverage. Section 5.5 describes the integration of special *security state* parameters into the `tester` and `generator` for the purpose of testing the security mechanisms. Considering the regression suite described in Section 3 as the basis, we now focus on the problem of testing security mechanisms for a given command in a given target state.

The number of possibilities per command is too large to be fully covered. A random sampling approach is therefore used for this problem as well.

5.3 Generation of a single credential

For a given command with given state parameters, a single *good credential* is generated using a constraint-satisfaction approach[3, 1] as follows:

1. Build a set of constraints which the credential fields should satisfy.
    - The constraints precisely define what values would make a good credential, as specified by the standard.
    - A single constraint may assert that some field must have a specific value (*e.g.*, the object-id in the credential must be same as the object-id field in the command). Alternatively, it may assert that a certain mask of bits should be set (*e.g.*, for a write command, the write permission bit must be set).
    - A single field may have several constraints, aggregated either by an AND relation or an OR relation. For example, 'the access tag field should be identical to the access tag attribute' OR 'it may be zero' (in which case it isn't tested).
2. Fill all fields with values that satisfy the constraints.
    - For a field that has several options, one option is randomly chosen and satisfied.
    - Each field is first filled with a 'minimal' value (*e.g.*, the minimum permission bits required for the command). Then it may be randomly modified within a range that is considered 'don't-care' by the constraint (*e.g.*, adding permission bits beyond the required ones).

Generating a *bad credential* starts by generating a good credential as described above. We then randomly select a single field in the credential and randomly 'ruin' its content so that it no longer satisfies its constraints.

How is this generation technique integrated with the existing tester? The `tester` controls whether a good or a bad credential is generated for a given command, and it expects commands with bad credentials to fail; that is, if they are completed successfully by the target, it is considered an error. Commands with good credentials are expected to behave as if security mechanisms do not exist.

When generating a bad credential, our `tester` generates one invalid field at a time. This is justified since almost all causes for rejecting a command are based on a single field. The standard does however specify that some fields should be validated before others. Since we only generate one invalid field at a time, this specification is not tested in our scheme.

Randomness is implemented as pseudo-randomness. This allows the `tester` to control the seed being used for the pseudo-random generation, enabling us to reproduce any encountered bug.

We now describe how to generate multiple random credentials for each command. This is required in order to cover as many rules as possible involved with validating a command.

5.4 Generation of many credentials

For a given command, the `tester` has several modes for generating credentials:

- A **deterministic mode**, where minimal credentials are generated.
- A **normal mode**, where each command is sent once with a random good credential. This mode allows testing many good credentials while activating the regression suite for other purposes.
- A **security-testing** mode described below.

The **security-testing** mode sends each command $2N$ times, N times carrying a random good credential and N times carrying a random bad credential. However, since OSD commands are not idempotent, this should be done carefully due to the following difficulties:

1. Each script command may depend on successful completion of previous commands. Hence, it is desirable to generate a command (whether with a good or a bad credential) only after all previous commands completed successfully.
2. On the other hand, some commands cannot be executed after they were already executed once (*e.g.*, creating an object). Hence, a command should not be re-sent after it was already sent with a good credential.

A script is executed in the **security-testing** mode in two stages, while each command is sent multiple times. The underlying assumption is that every script ends by 'cleaning up' all modifications it made, thus restoring the target to its old state.

> *First Phase:* each command in the script is sent N times with N randomly generated *bad* credentials, expecting N rejections. The command is sent once again, this time with a *good* credential, expected to succeed.
>
> *Second Phase:* the script is executed $N-1$ times in the normal mode (carrying a random good credential for each command), thus completing $N-1$ more good credentials for each command.

This technique proved to be very practical, mainly when used with long automatically-generated scripts. Its main contribution was in testing multiple bad paths. For example, a command accessing a non-existing object using an invalid credential or a command reading past object length while also accessing attributes that are not permitted by the credential.

5.5 Generating security-states at the target

The OSD security model defines a non-trivial mechanism for the *management of secret keys*. It involves interaction between the OSD target and a stateful security manager. To test this mechanism, the testing infrastructure should enable the generation of scenarios such as:

- Bad synchronization of key values between the security manager and the target.
- A client uses an old credential that was calculated with a key that is no longer valid.

To generate such scenarios, we let the *tester* act as a security manager and maintain a *local-state* of key values shared with the target. In addition to the regular `set_key` command we introduced two special script commands: `set_key_local`, `set_key_target` used to simulate scenarios where the key is updated only on one side. These extensions are within the black-box testing paradigm and do not require any modification of the target. These two commands proved very handy: by replacing normal `set_key` commands with these special commands, scripts with good key management scenarios can be easily transformed into scripts that simulate bad key management scenarios.

Another security mechanism requiring a state-aware tester is *revocation*. The OSD security model offers a per-object revocation mechanism via a special object attribute called the *policy/access tag*. By modifying this attribute, a policy manager may revoke all existing credentials allowing access to this object. Our `tester` was extended to support this mechanism by keeping the state of the policy/access tags for selected objects; this allowed us to verify an implementation of the revocation mechanism. By introducing a simplified keys-state and the revocation-attribute in the `script-generator`, we enabled the generation of many revocation scenarios as well as many key-management scenarios.

6 Benchmarks

Once an OSD is built, a natural question to ask is what is its performance. Or rather, how well does it perform? Benchmarking is a complex issue. There are many benchmarks for file systems and block disks; both close cousins to object-disks. However, we argue that these benchmarks are not adequate for measuring OSD performance.

Block-disk benchmarks contain only read and write commands while OSDs support a much richer command set. In the file system world, NFS-servers and NAS-boxes are the closest to object-disk. However, NFS benchmarks such as Spec-SFS [12] contain a lot of directory operations that are not supported by OSDs. Furthermore, the workload on a NAS-box is quite different from the expected workload for an object-disk. The following are a couple of comparison points:

1. In Spec-SFS, the NFS lookup operation takes up 27% of the workload. An OSD does not support an operation similar to lookup.
2. Some architectures place the RAID function above OSDs. This means that OSDs will contain file-parts and see read-modify-write RAID

transactions. This bears very little similarity to NFS style workloads.

3. An OSD supports a rich set of operations that are unique to

   OSDs and do not translate directly to file system operations. A good example is an OSD Collection. OSDs support an 'add to collection' operation. One possible use for this function is to add objects to an *in-danger* list, which counts objects that may need to be recovered in the event of file system failure. This operation affects the workload and it is not clear what weight it should be given in a benchmark.

The OSD workload is dependent on the file system architecture with which it is used. Therefore, if one sets out to build a Spec-SFS like benchmark for OSDs there are a lot of question marks around the choice of operation weights. As OSD-based file-systems are in their infancy, we expect the 'right' choice of operation weights to converge as the field matures and more OSDs emerge. As part of our OSD code, we built an initial framework for OSD benchmarks. We expect these benchmarks to be used as a tool to evaluate strengths/weaknesses of a specific OSD implementation, and to be used to design an OSD application on top of it. In the future, there will undoubtedly be a need to develop a 'common criteria' that can be used to compared and evaluate standard OSDs, very much like file systems implementations.

6.1 OSD benchmark suite

We developed a skeleton for a benchmark suite, which we believe can be extended and tuned in the future. Our suite is composed of two types: synthetic and spec-SFS like. All benchmarks are written as client executables, using the OSD initiator asynchronous API. Currently, they measure throughput and latency on the entire I/O path, but other statistic information can be gathered as needed.

- The `synthetic benchmarks` are built to test specific hand-made scenarios. They are useful for isolating and then analyzing a particular property of the system such as locking, caching, or fragmentation. Currently, the synthetic benchmarks consider the case of many small objects, or alternatively a single large object (similar to approach taken in [10]). Basic measurements include throughput and latency of read, allocating-write and non-allocating (re-)write commands as a function of the I/O size (ranging from 4K to 64K).

- The `Spec-sfs-like benchmarks` create, as pre-test stage, a large number of objects, and select a small part of it as a working set. The benchmark then chooses a command from an underlying distribution, randomly picks an object from the working set, selects arguments for the command from a given distribution, and initiates the command. Statistics are gathered on a per-command-type basis.

Below we provide a few examples of benchmark results that helped identify a problem, a weakness, or a bug in our system. Currently, this is the main use of the benchmark tool in our system.

6.2 Benchmark examples

Parameter tuning

The example in Figure 2 depicts throughput performance (in Mbytes/sec) of our OSD implementation. The data was obtained from one of the synthetic benchmarks for all-cache-hit reads, allocating-writes, and non allocating-write commands as a function of the I/O size. Our goal was to reach the maximum raw TCP performance over a 1Gb/sec network. In general, the throughput grows as a function of the I/O size. However, Figure 2 shows irregular behavior for writes (but not for reads) when I/O size is 32K. This called for closer analysis of a write command, which requires multiple requests-to-transfer (R2T) messages of various lengths. These R2T parameters need to be tuned to eliminate the observed irregularity.

Target behavior

The next example is taken from the Spec-sfs like benchmarks. It considers only read and write commands (all other weights are set to zero), with a uni-

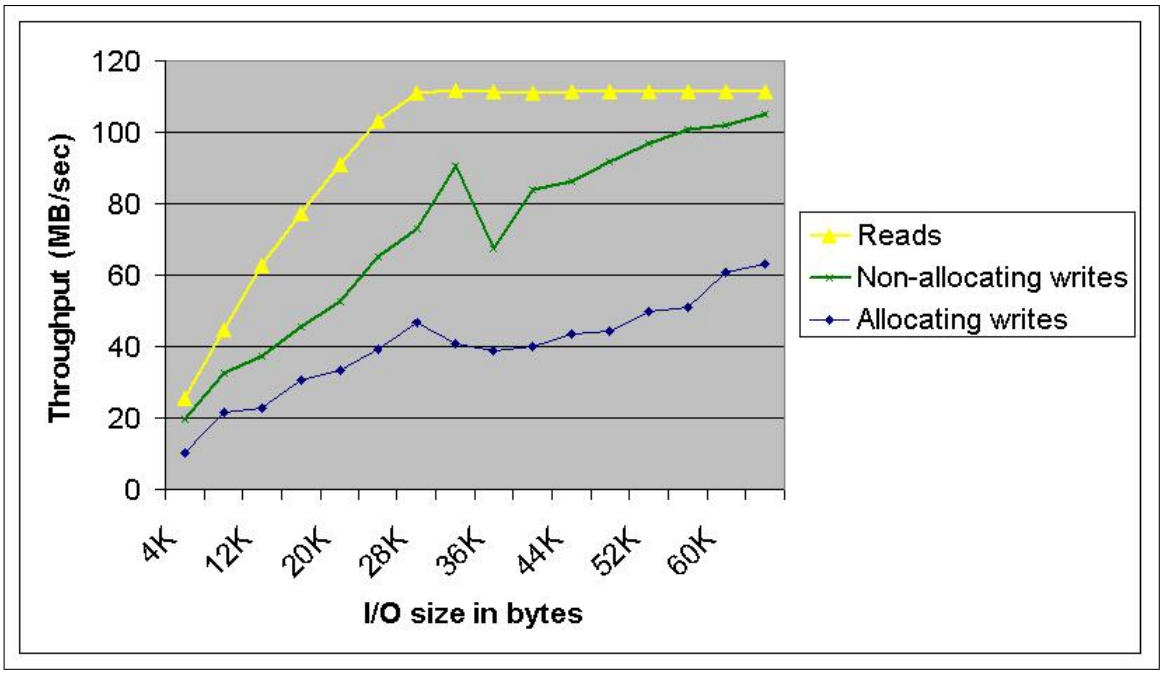

Figure 2: Throughput performance for all-cache-hits reads and writes (in Mbytes/sec as a function of the I/O size); irregularity is observed for writes at 32K

form size of 64K. Latency statistics for reads are depicted in Figure 3. When reads are all cache-hits, latency per command is distributed uniformly around 5 - 20 msec. However, in the example below, a bimodal behavior is observed with two very distinct means, indicating a mixture of cache-hit reads as well as cache-misses.

Identifying Bugs

The third example shows how we traced a bug in the Linux SCSI system using the benchmarks framework. As we ran longer benchmarks and plotted maximum command latency, we observed that there are always a small (statistically negligible) number of commands whose latency is substantially larger than the tail of the distribution. This indicated starvation in the system. Indeed, by looking closely we found that in the Linux SCSI implementation[7], SCSI commands are submitted in LIFO instead of FIFO order, without avoiding starvation (via a timeout mechanism for example)[2]. Because our benchmarks are designed not to leave the target idle, the LIFO behavior caused the said starvation. As a result, we patched the Linux kernel to support the appropriate ordering of OSD commands.

7 Related Work

Model-checking is a method that lies within the realm of black-box testing. This is because the target OSD code is not available to the tester. However, a model-checking approach can be very powerful, as shown in [15]. Proof systems can also be used to verify an implementation [2].

File-system debugging using comparison is employed in NFS-tee [13]. NFS-tee allows testing an NFSv3 server against a reference implementation by situating a proxy in front of the two systems. A workload is executed against the two systems and their responses are compared; a mismatch normally means a bug. This approach is similar to ours, however, NFS-tee does not combine any of the snapshot

[2]This behavior is documented in the Linux code in scsi_lib.c. Commands are placed at the head of the queue to support the scsi_device_quiesce function. Apparently, this has not been a problem in most systems since they do not overload the SCSI midlayer.

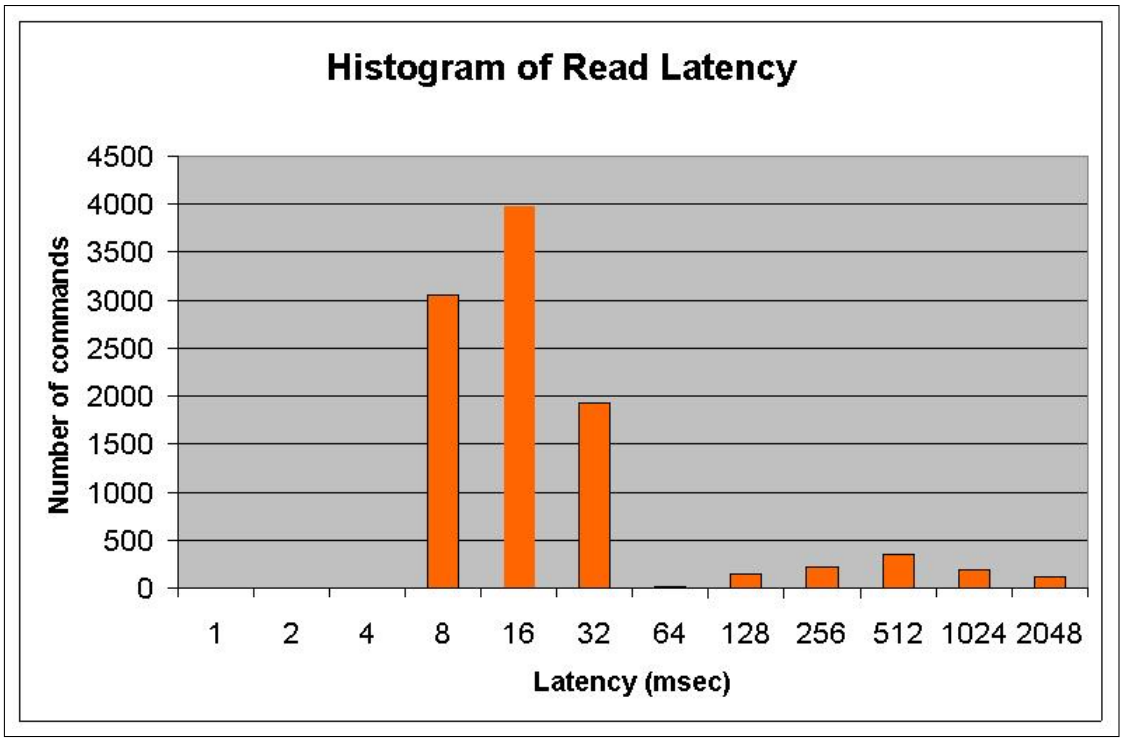

Figure 3: Latency of read commands (in msecs); a bimodal distribution is observed due to a mixture of cache hits and cache-misses

or gray-box techniques we employ.

There are many file-system testing and compliance suites (among the popular ones are [6, 8]); in fact, these are too numerous to list here. Most suites do not check file system recovery.

8 Summary and Future Extensions

In this paper, we report on our extensive efforts in building a comprehensive testing suite for T10-compliant OSDs, and our initial work on developing a common criteria for evaluating them. Object stores are new, and to-date there are only a handful of implementations. As the technology emerges, the need for such tools will be apparent. To the best of our knowledge, our work is the first attempt to address this need.

We report on the unique characterization of standard OSDs that made the testing procedure different and challenging, and show how we addressed these issues. Further work is required as more experience with building OSDs is gained, including:

- Improve testing coverage by enhancing the script-generation to address non-determinism beyond what is currently supported.
- Extend the script language to define broader recursive scripts, thus exploiting more complicated patterns of parallelism.
- Testing the other T10 security method (CMDRSP and ALLDATA).
- Testing advanced functionalities in the OSD T10 standard, *e.g.*, Collections.
- Enrich the benchmarks with real use-case data.

Acknowledgments

Efforts related to testing our object store implementation have been going on almost from day one in the IBM Haifa ObjectStone team. Many people in the team contributed ideas and work to this mission throughout the years. Special thanks to Guy Laden who wrote the first script generator, Grisha

Chockler who wrote the first version of the tester and Itai Segall who conceived the benchmarks suite. Avishay Traeger's input during the writeup of this paper was useful. Thanks to our colleagues from the IBM Haifa verification team: Roy Emek and Michael Veksler. Finally, thanks to Stuart Brodsky, Sami Iren and Dan Messinger from Seagate who used our tester and helped design testing for the CAPKEY security method.

References

[1] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek. Test program generation for functional verification of powerpc processors in ibm. In *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 279–285, New York, NY, USA, 1995. ACM Press.

[2] K. Arkoudas, K. Zee, V. Kuncak, and M. Rinard. Verifying a File System Implementation. In *Proceedings of the Sixth International Conference on Formal Engineering Methods (ICFEM 2004)*, 2004.

[3] R. A. DeMillo and A. J. Offutt. Constraint-based test data generation. *IEEE Transactions on Software Engineering*, 17(9), september 1991.

[4] *A Demonstration of an OSD-based File System, Storage Networking World Conference, Spring 2005*, April 2005.

[5] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. Object storage: The future building block for storage systems. a position paper. In *Proceedings of the 2nd International IEEE Symposium on Mass Storage Systems and Technologies, Sardinia Italy.*, pages 119–123, June 2005.

[6] *IOZone Filesystem Benchmark*. http://www.iozone.org/.

[7] *The Linux 2.6.10 SCSI Mid-layer Implementation, scsi_do_req API*.

[8] NetApp. *The PostMark Benchmark*. http://www.netapp.com/tech_library/3022.html.

[9] *A T10 iSCSI OSD Initiator*. http://sourceforge.net/projects/osd-initiator.

[10] M. I. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. N. Padmanabhan. File system logging versus clustering: A performance comparison. In *USENIX Winter*, pages 249–264, 1995.

[11] SNIA - Storage Networking Industry Association. *OSD: Object Based Storage Devices Technical Work Group*. http://www.snia.org/tech_activities/workgroups/osd/.

[12] Standard Performance Evaluation Corporation. *SPEC SFS97_R1 V3.0 Benchmarks*, August 2004. http://www.spec.org/sfs97r1.

[13] Y.-L. Tan, T. Wong, J. D. Strunk, and G. R. Ganger. Comparison-based File Server Verification. In *USENIX 2005 Annual Technical Conference*, April 2005.

[14] R. O. Weber. *SCSI Object-Based Storage Device Commands (OSD), Document Number: ANSI/INCITS 400-2004*. InterNational Committee for Information Technology Standards (formerly NCITS), December 2004. http://www.t10.org/drafts.htm.

[15] J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI*, pages 273–288, 2004.