blockbench: A Block-Based Load Generator and Benchmark Toolset

August 14, 2013

Abstract

blockbench is a set of tools to generate workloads for benchmarking block-based storage systems. The tool set supports a range of timing options for issuing I/Os, control over the read/write ratio, and the ability to set both the compression level and deduplication level of the written data, as well as other features. It is the ability to finely control timing and data content of the written blocks that makes this toolset more relevant for modern flash-based storage systems than existing tools, which often assume that the block-based storage system won't perform compression or deduplication.

1 Introduction

2 Building the Tools

The toolset uses autoconf and libtool to build the binaries. Building on Linux should be straightforward:

```
./configure
make
```

However, the tools require the sg3_utils include files, which may not be installed on your system. You can either install them as root, making them available to everyone, or install them in your home directory. If you do the latter, you'll probably need to add C preprocessor flags so that the build system can find the include files:

```
./configure CPPFLAGS='/users/me/include'
```

You can get a copy of sg3_utils at http://sg.danny.cz/sg/sg3_utils.html, or you can install it using standard installation tools on many recent Linux distributions.

3 Using the Tools

There are three steps in using blockbench on a system:

1. Set up the environment.
2. Prefill data into the part of the device(s) on which benchmarks will run.
3. Run the actual performance tests.

3.1 Setting up the Test Environment

`blockbench` can be used in any environment that supports block-based I/O, as long as it also supports "direct" I/O that bypasses the operating system's caches. We bypass the cache because doing so results in more accurate performance numbers: if I/O is done to the cache rather than the device, you can get very good results until you run out of cache space.

Since there are many different environments in which `blockbench` can be used, it's impossible to list all the ways in which the environment should be set up. However, it's important that there be one or more files corresponding to the raw devices on which `blockbench` will be used.

Typically, there's one raw device per file; however, it's certainly possible to have multiple devices corresponding to a single device. If this is done, care should be taken to ensure that multiple runs of `ioload` don't interfere with one another.

3.2 Prefilling Data

The tools in `blockbench` typically do random reads across the region being used in the test. To ensure that a random read returns "good" data, it's necessary to preload the region with data. Preloading data can also allow you to test the ability of the storage device to compress and deduplicate data by writing data patterns that are amenable to data reduction.

The `iofill` tool can be used to prefill a region with data. `iofill` allows you to control the type of data (compressibility) and the "space" of valid blocks, which determines the level of deduplication that might be available. It also allows you to control I/O size and the number of threads that run simultaneously (each individual thread waits for an outstanding I/O to complete before issuing the next one).

The data types supported by `iofill`, and the other `blockbench` tools are:

Abbrv	Data type
c	compressible data
h1 or **h**	data compressible to approx 1% of original size
h10	data compressible to approx 10% of original
h20	data compressible to approx 20% of original
h40	data compressible to approx 40% of original
e	email data from `.pst` files
v	virtual machine data (*default*)
u	uncompressible data

There are three types of options, test options (preceded by `-t`), module options (preceded by `-m`), and detail options (preceded by `-d`) that control the I/Os for which detailed information is reported. The test options control the number, size, and distribution of I/Os. A sample set of test options would be:

```
-t size=16384,nrep=1,skip=17
```

This tells `iofill` to use 16 KB I/Os, fill the region only once (the default), and skip 17 I/Os between requests. The `skip` parameter can be used to prevent `iofill` from writing data purely sequentially; some I/O systems are very good at prediction, and using a high value for `skip` makes this less useful. If the `skip` option is used, be aware that the entire region may not get filled if the interval is chosen poorly. For example, if the region is 1 GB and skip is 1 MB, the system will only write the blocks at 1 MB intervals.

The module options can be used to change the performance of `iofill` and the region that it uses; the data type is also specified here. A sample line might be:

```
-m threads=128,dtype=v,minoff=4,maxoff=4096,dedupblks=200
```

This would tell `iofill` to use 128 threads (maximum concurrency level of writes) to fill the region from 4 KB to 4096 KB. It would use data drawn from a predefined virtual memory data set, and would have a total

of 200,000 unique blocks in the data set. The `dedupblks` parameter controls the amount of deduplication; lower numbers for the size of the dedup set will result in more deduplication.

An overall command line for `iofill` might then be:

```
iofill -t size=16384 -m threads=128,dtype=v,minoff=4,maxoff=4096 /dev/vol1 /dev/vol
```

This would operate on both `/dev/vol1` and `/dev/vol2`.

3.3 Performance Testing

Now that the testing regions have been filled with data, we can start the performance testing. To do this, we'll use `ioload`, a tool that shares a lot of library code with `iofill`, but supports a lot more I/O patterns.

A sample run, with 75% read and 25% write, might use this command line:

```
ioload -n -o ioload-16k-rd75.out \
       -i pct=.75,size=16384,op=read \
       -i pct=.25,size=16384,op=write,dtype=v \
       -m initthreads=16,maxthreads=32,incrthreads=2,interval=10,maxlat=4000 \
       /dev/vol1 /dev/vol2
```

This command line would ignore I/O errors (`-n`), write output results to `ioload-16k-rd75.out`, and operate on `/dev/vol1` and `/dev/vol2`. Each of the `-i` lines is an I/O specifier. The first line specifies that 75% of the I/Os will be 16 KB reads, and the second says that 25% will be 16 KB writes, and that each write will generate data using the predefined VM data (see Section 3.2). The module-specific options say that the test will start with 16 threads, building by 2 threads every 10 seconds, to a total of 32 threads. The test will stop when the 90th percentile latency exceeds 4 ms (4000 μs). The system will run for 60 seconds before starting to report results, though `ioload` generates load during the warmup period, and external tools can measure performance during this time.