Chapter 13
Storage Benchmarks

Klaus-Dieter Lange, Don Capps, Sitsofe Wheeler, Sorin Faibish, Nick Principe, Mary Marquez, John Beckett, and Ken Cantrell

Many operating systems (OS) and information technology (IT) solutions have been tested and tuned for the storage subsystems to work well with frequently used applications. This accelerates the storage input/output (I/O) for the respective subsets of workloads. Nonetheless, the application operation mix will likely change over time as applications evolve, for example, the system administrator decides to allocate the same server/storage solution for additional office automation tasks.

Several benchmarks are available to evaluate the storage performance of a specific storage system or storage component. They can be used by system administrators to evaluate and compare different products and ensure high performance for their particular environment. This chapter presents a brief history of the SPEC System File Server (SFS) benchmarks and takes a closer look at SPEC SFS 2014. It then introduces the benchmarks from the Storage Performance Council (SPC) and the IOzone file system benchmark. Finally, the Flexible I/O Tester (fio) is presented, showing some examples of how it can be used to measure I/O performance.

13.1 Historical Perspective on System File Server Benchmarks

In the 1990s, commercial Network File System (NFS) storage server arrays started to become mainstream. No longer was storage the realm of government, academia, and large corporations. This increase of storage solution choices created a need for a benchmark to enable users of NFS servers to select the solution with the highest performance. NFS server vendors joined forces with academics and government to build an NFS benchmark with relevant and meaningful performance metrics. In October 1992, the synthetic benchmark LADDIS (Wittle and Keith, 1993), named using the initials of the involved organizations (Legato, Auspex, Data General, Digital Equipment, Interphase, and Sun Microsystems), was released. It was based on the *nhfsstone* workload and measured guaranteed performance, that is, performance achieved for a given latency target. A higher LADDIS score indicates higher I/O performance at a lower latency.

S. Kounev et al., *Systems Benchmarking*, https://doi.org/10.1007/978-3-030-41705-5_13

In 1993, the LADDIS group joined the Standard Performance Evaluation Corporation (SPEC) and became its System File Server (SFS) Subcommittee. There, the NFS benchmark was enhanced for the NFSv2 protocol and released under its new name SPEC SFS 93. The SPEC SFS 97 benchmark, released in December 1997, was further enhanced with new functionality and support for the NFSv3 protocol.[1] It became one of the most popular storage performance benchmarks during that time.

In June 2001, a series of bug fixes were released via SPEC SFS 97 V2.0. Later in 2001, SPEC released SPEC SFS 97_R1 V3.0 with additional bug fixes and support for Linux and FreeBSD as well as initial support for various UNIX operating systems. The benchmark remained an NFS benchmark; nonetheless, the need for a new benchmark that supports the Server Message Block (SMB) protocol, in particular the Common Internet File System (CIFS), was increasing as most storage vendors started to support both NFSv3 and SMB protocols. At this point, the members of the SFS Subcommittee started to develop a new benchmark for Windows servers using the SMB protocol. In 2008, the work on this benchmark was completed and the first dual protocol storage benchmark—SPEC SFS 2008—was published.[2] In addition to the introduction of the SMB protocol, several enhancements were made to the NFSv3 benchmark, including operation mix change and adding new metadata operations, aligned with the evolving requirements of the storage industry.

During the lifetime of SPEC SFS 2008, its user base started asking for support for measuring the performance of the clients and servers in a single unified benchmark. Coincidently, in December 2010, Don Capps was finishing his development of *Netmist*—the first benchmark and framework designed as a system benchmark that runs at the system call level instead of the protocol level. He granted SPEC the permission to use it as the basis for the next generation file server benchmark. Netmist combines benchmark ideas from both the SFS benchmarks and the IOzone benchmark (see Section 13.4), and it was designed as a multi-client, multi-server benchmark.

13.2 SPEC SFS 2014

After 4 years of joint development, SPEC SFS 2014 was released in November 2014.[3] It introduced many novel benchmark ideas inspired by the established file server benchmarks and included support for cluster file systems (e.g., Lustre and GPFS) as well as network file servers (e.g., based on NFSv3, NFSv4, and SMB). The SFS Subcommittee also implemented the support for local POSIX file systems created on block storage device benchmarks via any POSIX file systems on the raw block devices and supporting any type of client host OSes including SOLARIS, Linux, Windows, SGI, AIX, etc., and any client local POSIX file systems.

[1] NFS v3 protocol; IETF 1995: https://tools.ietf.org/html/rfc1813

[2] SPEC SFS 2008 benchmark: https://www.spec.org/sfs2008

[3] SPEC SFS 2014 benchmark: https://www.spec.org/sfs2014

The SPEC SFS 2014 benchmark introduced the concept of *business metrics (BMs)*, inspired from real storage applications, and added the capability to easily modify existing BMs and to create new BMs for research purposes. The five included BMs (see Table 13.1) measure guaranteed performance based on the same request–response principles of the five most popular types of storage application characteristics (e.g., mixes for metadata and data, read and write, and for different I/O sizes). With the new capability to saturate all physical resources, including CPU, disk, pipes (FC and IP), BUSes, and memory, the SFS benchmark evolved into an application benchmark. This enabled the different storage vendors to showcase their storage solutions for the BM that matched their customers' usage for either protocol.

Table 13.1: Workloads and their business metric names

Workload	Business metric name
Electronic design automation (EDA)	Job sets
Database (DATABASE)	Databases
Software build (SWBUILD)	Builds
Video data acquisition (VDA)	Streams
Virtual desktop infrastructure (VDI)	Desktops

In 2016, the SPEC SFS 2014 benchmark was enhanced to also serve as a load generator used for measuring the power consumption of storage servers as defined by the Storage Networking Industry Association (SNIA)—a feature used by SNIA in the Emerald specification as well as by the U.S. Environmental Protection Agency's (EPA) Energy Star program for storage certification.

All previous SFS benchmark results were presenting only two performance metrics, the NFS/CIFS I/O operations (IOPS) and overall response time (ORT), as well as a result table and a graph (see Table 13.2 and Figure 13.1[4]), showing the guaranteed performance achieved for each requested I/O load. Starting with SPEC SFS 2014, the new performance variables, Business Metric (workload specific) and Bandwidth in MB/sec, were added to the result (see Table 13.3 and Figure 13.2[5]).

With the continuous evolution of storage applications and technology, including new storage media like solid-state drives (SSD) and non-volatile memory (NVM), additional workloads become of interest and the current workloads need to be modified or replaced to reflect new users' needs and usage models of new application areas like machine learning, Genomics, and others. The SPEC OSG Storage Subcommittee, the new name of the SFS Subcommittee, is working to deliver the next generation of SFS benchmarks, addressing the need for new features and representative workloads for the storage industry.

[4] Corresp. result: https://www.spec.org/sfs2008/results/res2008q1/sfs2008-20080218-00083.html

[5] Corresp. result: https://www.spec.org/sfs2014/results/res2014q4/sfs2014-20141029-00002.html

Table 13.2: Exemplary SPEC SFS 2008 publication table

Throughput (ops/sec)	Response time (ms)
320	1.5
642	1.8
961	2.0
1,285	2.3
1,607	2.6
1,924	3.2
2,244	4.0
2,579	5.6
2,897	8.5
3,088	10.6

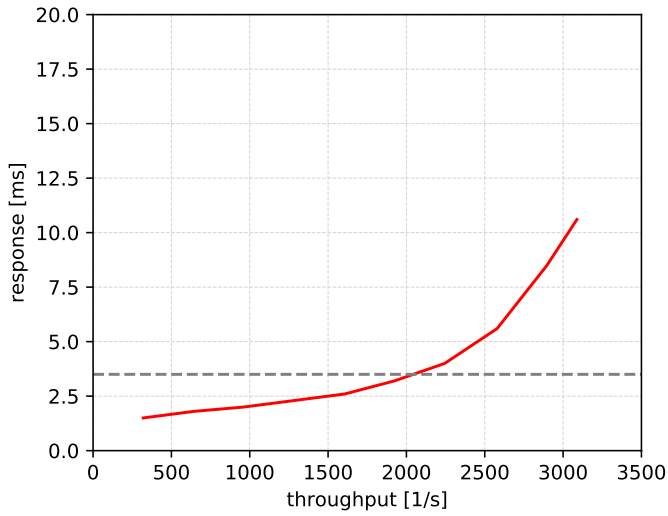

Fig. 13.1: Exemplary SPEC SFS 2008 publication graph

Table 13.3: Exemplary SPEC SFS 2014 publication table

Business metric (builds)	Average latency (ms)	Builds (ops/sec)	Builds (MB/sec)
2	0.6	1,000	12
4	0.7	2,000	25
6	0.7	3,000	38
8	0.7	4,000	51
10	1.0	5,000	64
12	1.1	6,000	77
14	1.1	7,000	90
16	1.0	8,001	103
18	0.9	9,000	116
20	1.0	10,001	128
22	1.1	11,001	141
24	1.3	12,001	154

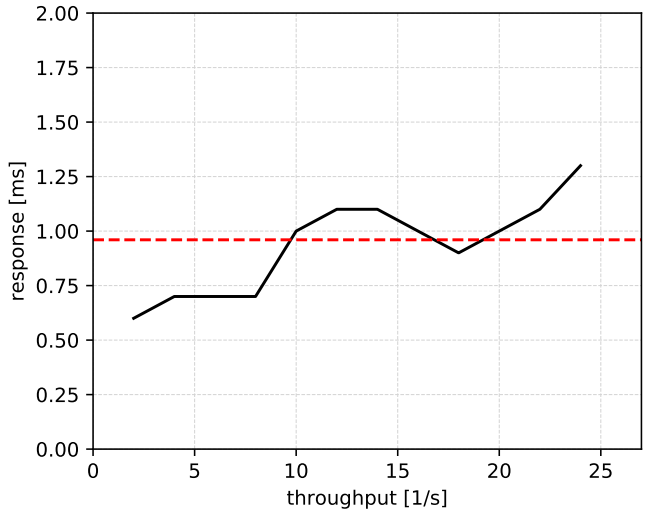

Fig. 13.2: Exemplary SPEC SFS 2014 publication graph

13.3 Storage Performance Council (SPC)

The vendor-neutral SPC was founded in 1998 with the goal to develop industry-standard benchmarks focusing on storage subsystems and to publish third-party audited benchmark results that include performance and pricing information. Their core benchmarks—SPC-1 and SPC-2—measure the performance of storage systems, and they utilize a common SPC framework for benchmark components.

13.3.1 SPC-1

Introduced in 2001, SPC-1 had a single workload and targeted storage performance of business-critical applications with a high random I/O mix and a series of performance hotspots. The benchmark includes query and update operations, and it covers a broad range of business functions, system configurations, and user profiles.

The SPC-1 benchmark uses the concept of *stimulus scaling units (SSUs)* to scale the I/O load while maintaining the operation mix and constraints. The balance between application I/O and logging I/O is maintained as the SSUs are scaled to the desired I/O load. Application storage units (ASUs) form the abstracted storage configuration, which provides the environment in which the workload (represented by SSUs) is executed. Each ASU is considered the source or destination of data that requires persistence beyond the benchmark run itself. Figure 13.3 shows an example of the distribution of the average response time for the first repeatability test run at the 100% load level.

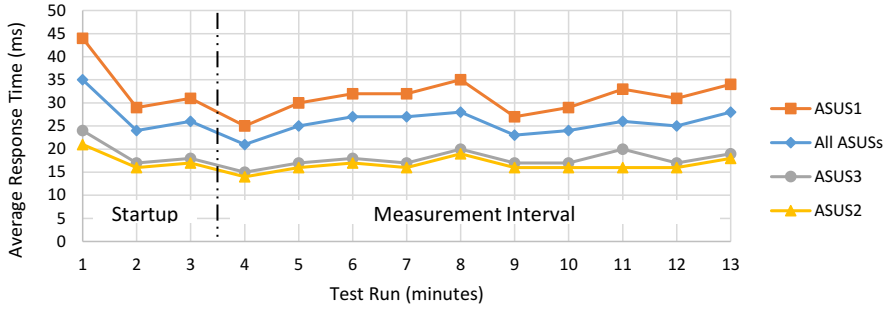

Fig. 13.3: SPC-1 average response time distribution

SPC-1 has several workload components scaled from smallest to largest:

- I/O REQUEST: A single unit of work,
- I/O STREAM: A single sequence of I/O REQUESTS,
- ASU STREAM: A collection of I/O STREAMs,
- WORKLOAD: A collection of ASU STREAMs.

The performance results and response time are part of the benchmark final report, which includes detailed system and storage subsystem configuration details as well as pricing information.

13.3.2 SPC-2

SPC-2, introduced in 2005, has three different workloads to stress the storage system with large-scale sequential data movement, which is one of several differences to the random I/O nature of the SPC-1 standard. The modeled I/O operations include large file processing, large database queries, and video on demand.

The SPC-2 benchmark leverages structured patterns of I/O requests referred to as *streams*; the number of concurrent streams varies during benchmark execution. Three or more of these streams are executed for each workload; the maximum and intermediate number of streams are defined by the benchmark tester. Figure 13.4 shows an example of the average data rate per stream for a load of 60,000 streams.

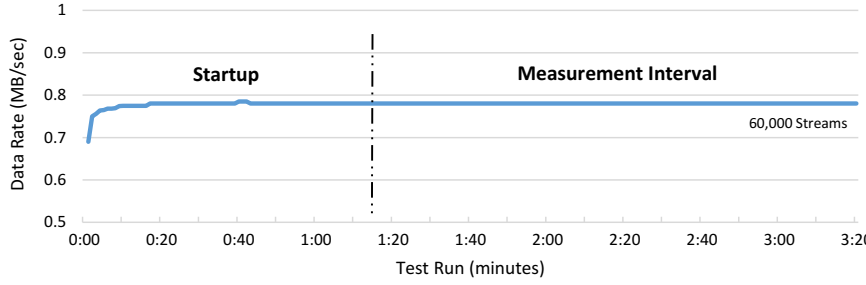

Fig. 13.4: SPC-2 average data rate per stream

13.3.3 Component and Energy Extension

The derived SPC-1C and SPC-2C benchmarks target specific storage components like storage devices and controllers, storage enclosures, and storage software. SPC-1C and SPC-2C retain the essential random or sequential nature, respectively, of their benchmark progenitors. These two benchmarks are intended to provide performance data for individual storage components as opposed to a larger storage configuration.

Each SPC benchmark has an optional energy extension (SPC-1/E, SPC-2/E, SPC-1C/E, and SPC-2C/E), which adds a mode of execution in which also the power consumption is measured. Power consumption is measured at three load intervals (idle, moderate, and heavy) and reported with the performance results and pricing

of the regular benchmark run. SPC benchmark results with energy extension include the following additional metrics:

- Nominal Operating Power (W): The average power consumption across the three intervals,
- Nominal Traffic in IOPS: The average I/O measured across the three intervals,
- Operating IOPS/watt: The computed power metric representing the overall efficiency for I/O traffic,
- Annual Energy Use (kWh): The estimated annual energy usage.

13.4 The IOzone Benchmark

IOzone was initially designed and written by William Norcott and released in the early 1980s. The initial version, a fairly simple C-code, measured the time for opening a file, write/read data, and close the file.

Don Capps started his work on extending IOzone's functionality in 1985; he fundamentally redefined IOzone for more accurate performance measurements of file systems. He added support for large-scale NUMA supercomputers in 1991 and expanded IOzone's capability to cover multiple file servers running in parallel.

In 2000, the IOzone.org site was created and the IOzone development continued under a freeware licensing model with Don Capps as the benchmark maintainer. This license model allows the users to compile and use the benchmark for free on any platform and OS. The IOzone benchmark continues to be a living project with contributions from developers worldwide (e.g., Android support for the use of IoT devices). Nonetheless, developers are not allowed to distribute changes by themselves, as it is maintained by a single entity to preserve the integrity of code contributions and their proper integration.

Similar to fio,[6] Iometer,[7] and IOR,[8] the IOzone benchmark has evolved to one of the more sophisticated file system performance benchmark utilities, generating and measuring a variety of file operations (see Table 13.4).

Table 13.4: IOzone's file operations

read	re-read	fread	random read	aio read	pread variants
write	re-write	fwrite	random write	aio write	pwrite variants
read backwards	read strided	mmap			

IOzone has been ported to many platforms and is available on most OSes including AIX, BSDI, HP-UX, IRIX, FreeBSD, Linux, OpenBSD, NetBSD, OSFV3, OSFV4,

[6] Flexible I/O Tester (fio): https://fio.readthedocs.io

[7] Iometer Project: http://www.iometer.org

[8] IOR Benchmark: https://media.readthedocs.org/pdf/ior/latest/ior.pdf

OSFV5, SCO OpenServer, Solaris, Mac OS X, and Windows (via the Cygwin runtime application[9]). Its results can be exported into useful graphs (e.g., Figure 13.5 depicts the fwrite performance under Windows), which can be leveraged to show performance characteristics and bottlenecks of the disk I/O subsystem, enabling users to optimize their applications to achieve the best performance for their platform and OS. This is one of the reasons why the benchmark is widely used to evaluate HPC storage for supercomputers and computer clusters.

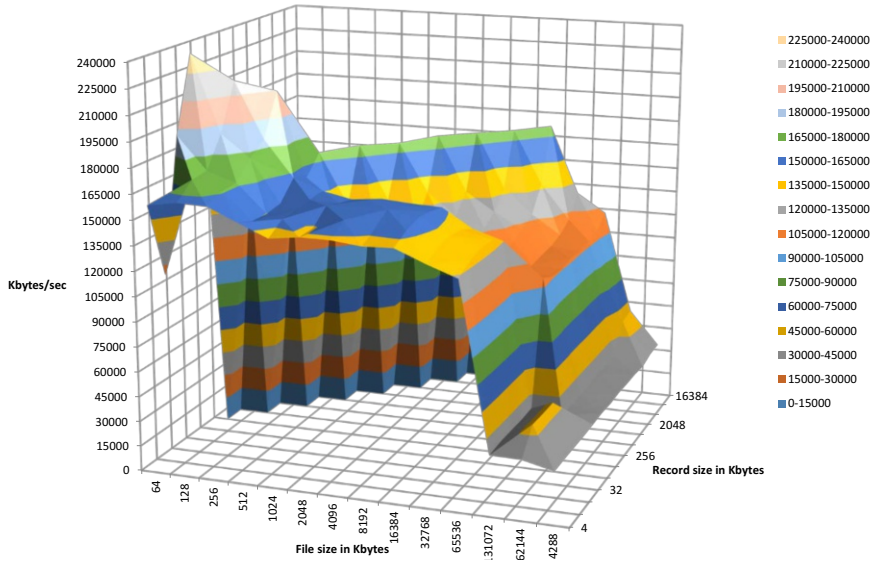

Fig. 13.5: IOzone fwrite performance

13.5 Flexible I/O Tester (fio)

The Flexible I/O Tester (fio) was designed by Jens Axboe in 2005, filling the void for a flexible method to simulate customizable I/O workloads and to gain meaningful I/O statistics on the Linux I/O subsystem and its schedulers.

The fio architecture is comprised of three major parts: (1) *front-end* that parses a job description file; (2) *back-end layer* that performs common work like managing parallel workers, collecting I/O statistics, and generating/validating I/O patterns; and (3) implementation of pluggable *ioengines* that send I/O in different ways over the network via library calls.

[9] Cygwin: http://www.cygwin.com

The small and portable fio code is pre-packaged by major Linux distributions because of its versatile nature of exploring various aspects of storage subsystems:

- Investigation of storage performance and root-cause analysis of bottlenecks,
- Modeling of workloads with a balanced read/write access mix across multiple workers,
- Replay of recorded and hand-built workload patterns,
- Analysis of the effectiveness of different caching algorithms, and
- Reproduction of hardware and software issues.

Fio has been ported to many platforms; nonetheless, its capabilities on other platforms might not be the same as on Linux, because some features might not have been ported, or different platforms may not implement the same functionality in the same way. The latest version can be found at the fio Git repository.[10] In the following, we present a series of examples illustrating fio's capabilities on Linux.

13.5.1 Running a Simple Job

A fio job file contains a set of statements describing what I/O workload should be executed. The following example describes a new job called `simple` that creates a file at the path `/tmp/fio.tmp` with a size of two megabytes (by default, all single- and three-letter storage units in fio are powers of two, for example, 2 M is 2,097,152 bytes). It then performs read I/O using the default ioengine (on Linux, this is psync) and the default block size (4,096 bytes).

```
[simple]
filename=/tmp/fio.tmp
size=2M
rw=read
```

If the above was saved to the file `simple.fio`, it can be run via:

```
fio simple.fio
```

Running this job will create an output similar to the following (lines 1–31):

```
1  simple: (g=0): rw=read, bs=(R) 4096B-4096B, (W) 4096B-4096B,
       (T) 4096B-4096B, ioengine=psync, iodepth=1
2  fio-3.16
3  Starting 1 process
4  simple: Laying out IO file (1 file / 2MiB)
5
6  simple: (groupid=0, jobs=1): err= 0: pid=19566: Sat Nov 9
       11:39:05 2019
7    read: IOPS=56.9k, BW=222MiB/s (233MB/s)(2048KiB/9msec)
8      clat (nsec): min=896, max=944834, avg=16216.53,
           stdev=101996.63
```

[10] Flexible I/O Tester (fio) Git repository: https://github.com/axboe/fio.git

```
9       lat (nsec): min=934, max=944898, avg=16284.52,
            stdev=102006.39
10      clat percentiles (nsec):
11       |  1.00th=[   940],  5.00th=[  1032], 10.00th=[  1064],
             20.00th=[  1688],
12       | 30.00th=[  1784], 40.00th=[  1800], 50.00th=[  1816],
             60.00th=[  1832],
13       | 70.00th=[  1848], 80.00th=[  1880], 90.00th=[  1960],
             95.00th=[  2160],
14       | 99.00th=[716800], 99.50th=[872448], 99.90th=[946176],
             99.95th=[946176],
15       | 99.99th=[]
16    lat (nsec)   : 1000=2.15%
17    lat (usec)   : 2=90.23%, 4=4.88%, 20=0.39%, 50=0.20%,
          250=0.20%
18    lat (usec)   : 500=0.39%, 750=0.78%, 1000=0.78%
19    cpu          : usr=0.00%, sys=25.00%, ctx=13, majf=0, minf=10
20    IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%,
          32=0.0%, >=64=0.0%
21       submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%,
             64=0.0%, >=64=0.0%
22       complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%,
             64=0.0%, >=64=0.0%
23       issued rwts: total=512,0,0,0 short=0,0,0,0 dropped=0,0,0,0
24       latency   : target=0, window=0, percentile=100.00%, depth=1
25
26  Run status group 0 (all jobs):
27     READ: bw=222MiB/s (233MB/s), 222MiB/s-222MiB/s
           (233MB/s-233MB/s), io=2048KiB (2097kB), run=9-9msec
28
29  Disk stats (read/write):
30      dm-0: ios=0/0, merge=0/0, ticks=0/0, in_queue=0,
           util=0.00%, aggrios=12/0, aggrmerge=0/0, aggrticks=8/0,
           aggrin_queue=8, aggrutil=3.31%
31    sda: ios=12/0, merge=0/0, ticks=8/0, in_queue=8, util=3.31%
```

The output is comprised of the following parts:

Line 1:	A summary of some of the parameters within the job
Line 2:	The fio version
Lines 3–4:	Information about the job starting
Lines 5–6:	Process identification
Line 7:	Average IOPS and bandwidth information
Lines 8–15:	Latency break down per I/Os
Lines 16–24:	Further breakdown of the I/O information
Lines 25–27:	Summary of I/O by group
Lines 28–31:	Information about how the kernel performed disk I/O

The key information on how the job performed (lines 7–15) is depicted in Figure 13.6; detailed guidelines on the interpretation of the different parts of the output can be found in the fio documentation.[11]

[11] https://fio.readthedocs.io/en/latest/fio_doc.html#interpreting-the-output

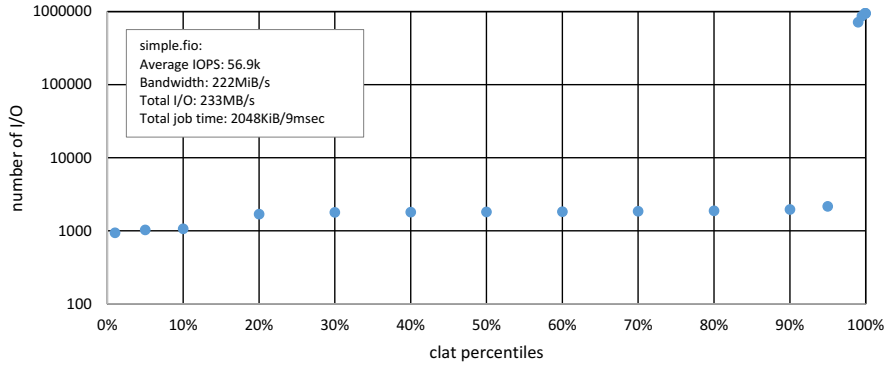

Fig. 13.6: Completion latency (clat) per I/O

A job can also be specified using command line options; for example, the previous job can be written as:

```
fio --name=simple --filename=/tmp/fio.tmp --size=2M --rw=read
```

A section is started by using the `-name` option followed by a value, and parameters become double-dashed options followed by their value. Although job files are useful for repeatability and sharing, the remaining jobs shown in this chapter are specified as command line options for the sake of brevity.

In the read job above, fio actually wrote the data to the file semi-randomly before reading it back. This is done to prevent special-case optimizations (which may be applied within the storage stack) from distorting the results of a particular run. An in-depth explanation on how the random data is generated can be found in the fio documentation.[12]

13.5.2 More Complex Workloads

It is easy to change the simple workload from performing read access to write access by basically changing the `-rw` command from `read` to `write`. It should be pointed out that using write workloads will destroy the data in the files specified. The following example shows a job that performs sequential writes with an increased block size of 64 kilobytes (64×1024^2 bytes):

```
fio --name=simplewrite --filename=/tmp/fio.tmp --size=2M
    --rw=write --bs=64k
```

Fio has the capability to work with block devices directly, allowing one to measure performance without the overhead of the file system. In the following examples,

[12] https://fio.readthedocs.io/en/latest/fio_doc.html#buffers-and-memory

/dev/sdd represents such a block device, and fio will try to write to 64 kilobyte sized blocks in a random order (-rw=randwrite), but it will cover each block exactly once:

```
fio --name=simplewrite --filename=/dev/sdd --rw=randwrite
    --bs=64k
```

Additionally, while fio will try and flush kernel caches on supported platforms before starting a job, by default, no flushing takes place when the job finishes; thus, data may still be in kernel RAM caches (and non-volatile disk caches). The end_fsync=1 option can be used to ensure that write data has reached the disk by the time the job finishes.

```
fio --name=simplewrite --filename=/dev/sdd --rw=randwrite
    --bs=64k --end_fsync=1
```

Fio jobs can be run in parallel in order to model real-life environments with multiple concurrent workloads. The following simple example runs two read workloads accessing different files in parallel:

```
fio --name=simple1 --filename=/tmp/fio1.tmp --size=2M --rw=read
    --name=simple2 \
    --filename=/tmp/fio2.tmp --size=2M --rw=read
```

A global section can be utilized to share common parameters between the jobs, eliminating duplications.

```
fio --size=2M --rw=read --name=simple1 --filename=/tmp/fio1.tmp
    --name=simple2 \
    --filename=/tmp/fio2.tmp
```

By default, all jobs are part of the same group, which allows fio to provide a way of summarizing some of the results of multiple jobs (see lines 26–27 in the previous output example). Note that this summary information may be inaccurate if the jobs do not actually start at the same time.

Fio provides a number of options for modeling simultaneous reads and writes. For cases where reads and writes are independent of each other, the following method can be applied:

```
fio --size=2M --filename=/tmp/fio1.tmp --name=writes --rw=write
    --name=read --rw=read
```

If they are somehow dependent on each other, the reads and writes can be handled by the same job via the -rw=readwrite option, and the mix can be specified via the -rwmixread parameter. The following example requests four reads for every write:

```
fio --size=2M --filename=/tmp/fio1.tmp --name=mix --rw=readwrite
    --rwmixread=80
```

13.5.3 Unusual I/O Patterns

The previous job examples perform uniformly distributed random I/O across the area being accessed. In some cases, for example, when analyzing the effectiveness of caching, it might be helpful to access different parts of the targeted area with different frequency. Fio has multiple ways to define such a distribution; Figure 13.7 shows an example `cache_test` that utilizes the `-random_distribution=zoned` option:

```
fio --name=cache_test --filename=/tmp/fio1.tmp --size=20G
    --rw=randread \
    --random_distribution=zoned:30/15:14/15:40/5:14/20:2/45
```

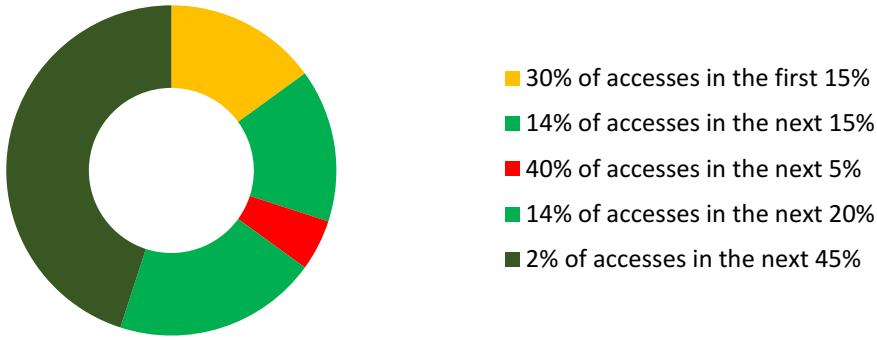

Fig. 13.7: Randomly distributed I/O via zones

Many unusual I/O patterns can be created via the vast possibilities of combining different fio options. The last two examples in this chapter might be helpful in order to recreate I/O patterns to root-cause hardware and software issues of extent-based storage.

The `gappy` job writes every other 8 kilobytes of `/tmp/fio1.tmp`:

```
fio --name=gappy --filename=/tmp/fio1.tmp --size=2M
    --rw=write:8k --bs=8k
```

The `backwards` job seeks 16 kilobytes backwards after every 8 kilobyte writes are done, before writing the next 8 kilobytes.

```
fio --name=backwards --filename=/tmp/fio1.tmp --size=2M
    --rw=write:-16k --bs=8k
```

13.5.4 ioengines

The ioengine used in the above examples has been synchronous, which means that fio will wait for an I/O operation to complete before sending another I/O operation. However, modern disks and disk controllers have multiple queues that achieve maximum performance when many I/O operations are submitted in parallel. In cases where a kernel cache is being used, the kernel's buffering can help synthesize that parallelism at a small cost. However, some I/O engines can create that asynchrony themselves with lower overhead.

On Linux, the `libaio` ioengine is typically used for this purpose, but it comes with strict requirements to prevent blocked submissions:

- I/O must be sent using the O_DIRECT option.
- The amount of I/O backlog should be kept limited.

It is important to adhere to these rules and avoid blocking submissions, because fio will not be able to queue any more I/O until submissions return.

The following example job utilizes the `libaio` engine and sets the `iodepth`, which controls the maximum amount of I/O operations to be sent simultaneously and queued. There is no guarantee that, at any given point, the `iodepth` amount of I/O operations will be queued up, as I/O operations are queued one at a time, and if their completion is fast enough, there will not be much outstanding work at any given time.

```
fio --ioengine=libaio --iodepth=32 --name=parallelwrite
    --filename=/dev/sdd \
    --rw=randwrite --bs=64k
```

It is quite common to run a workload for a fixed amount of time in order to reproduce hardware or software issues. This can be achieved by utilizing the `time_based` and `runtime` options. The following fio job will continue to loop the pattern until `runtime` has expired:

```
fio --name=one-minute --filename=/tmp/fio1.tmp --size=2M
    --rw=write --time_based \
    --runtime=1m
```

In order to measure the maximum performance of very fast storage subsystems, it might be necessary to minimize fio's overhead. The `io_uring`[13] interface was introduced with the 5.1 Linux kernel, and it is supported by fio version 3.13 and higher. It has a lower overhead and can therefore push higher bandwidths than the previous libaio/KAIO interface. Using it is just a matter of changing the ioengine:

```
fio --ioengine=io_uring --iodepth=32 --name=parallelwrite
    --filename=/dev/sdd \
    --rw=randwrite --bs=64k
```

[13] https://kernel.dk/io_uring.pdf

13.5.5 Future Challenges

Driven by an active development community, fio has grown to be a popular tool, continuously offering new features, bug fixes, ioengines, and support for new platforms. In this chapter, we touched upon some of the many capabilities of fio, which should help guide investigation in storage performance, root-cause analysis of bottlenecks, and the reproduction of hardware and software issues. A future area to explore would be the ability to create generative models based on the analysis of previously recorded I/O traces. This would enable the portability of realistic workload replays.

13.6 Concluding Remarks

Several benchmarks have emerged in the last decades specifically designed to evaluate the performance of storage systems and storage components. This chapter presented a brief history of the SPEC System File Server (SFS) benchmarks and took a closer look at SPEC SFS 2014. It then introduced the benchmarks from the Storage Performance Council (SPC) and the IOzone file system benchmark. Finally, the Flexible I/O Tester (fio) was presented, showing some examples of how it can be used to measure I/O performance.

With the continuous evolution of storage applications and technology, including new storage media like solid-state drives (SSD) and non-volatile memory (NVM), additional storage workloads become of interest and the current workloads need to be modified or replaced to reflect new users' needs and usage models of new application areas like machine learning, Genomics, and others.

References

Wittle, M. and Keith, B. E. (1993). "LADDIS: The Next Generation in NFS File Server Benchmarking". In: *Proceedings of the 1993 Summer USENIX Technical Conference*. (Cincinnati, Ohio). USENIX Association: Berkeley, CA, USA, pp. 111–128 (cited on p. 285).