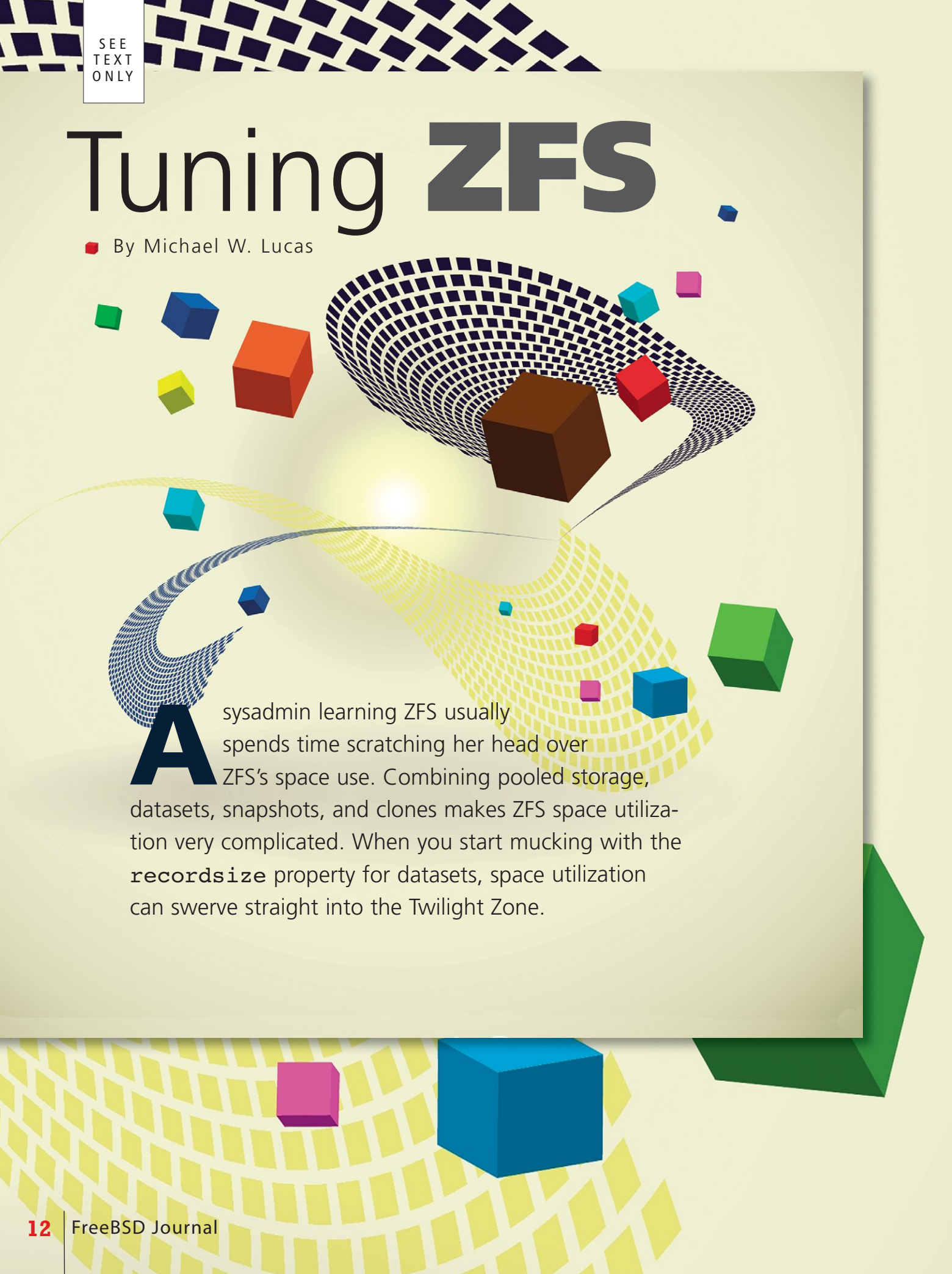


Tuning ZFS

By Michael W. Lucas

The background features a complex 3D graphic. It includes several colorful cubes (blue, orange, green, red, brown, cyan, pink) scattered across the scene. Two prominent grid-like structures, one blue and one yellow, curve through the space, resembling funnels or orbits. A bright light source is visible in the center, casting a glow. The overall aesthetic is clean and modern, with a focus on geometric shapes and perspective.

A sysadmin learning ZFS usually spends time scratching her head over ZFS's space use. Combining pooled storage, datasets, snapshots, and clones makes ZFS space utilization very complicated. When you start mucking with the `recordsize` property for datasets, space utilization can swerve straight into the Twilight Zone.

The `recordsize` property gives the maximum size of a logical block in a ZFS filesystem dataset. The default `recordsize` is 128 KB, which comes to 32 sectors on a disk with 4 KB sectors, or 256 sectors on a disk with 512 byte sectors. The maximum record size was increased to 1 MB with the introduction of the `large_blocks` feature flag in 2015. Many database engines prefer smaller blocks, such as 4 KB or 8 KB. It makes sense to change the `recordsize` on datasets dedicated to such files. Even if you don't change the `recordsize`, ZFS automatically sizes records as needed. Writing a 16 KB file should take up only 16 KB of space (plus metadata and redundancy space), not waste an entire 128 KB record.

This comes most into play with databases.

Databases and ZFS

Many ZFS features are highly advantageous for databases. Every DBA wants fast, easy, and efficient replication, snapshots, clones, tunable caches, and pooled storage. While ZFS is designed as a general-purpose filesystem, you can tune it to make your databases fly.

Databases usually consist of more than one type of file, and since each has different characteristics and usage patterns, each requires different tuning. We'll discuss MySQL and PostgreSQL in particular, but the principles apply to any database software.

The most important tuning you can perform for a database is the dataset block size—through the `recordsize` property. The ZFS `recordsize` for any file that might be overwritten needs to match the block size used by the application.

Tuning the block size also avoids write amplification. Write amplification happens when changing a small amount of data requires writing a large amount of data. Suppose you must change 8 KB in the middle of a 128 KB block. ZFS must read the 128 KB, modify 8 KB somewhere in it, calculate a new checksum, and write the new 128 KB block. ZFS is a copy-on-write filesystem, so it would wind up writing a whole new 128 KB block just to change that 8 KB. You don't want that. Now multiply this by the number of writes your database makes. Write amplification eviscerates performance.

While this sort of optimization isn't necessary for many of us, for a high-performance system it might be invaluable. It can also affect the life of SSDs and other flash-based storage that can handle a limited volume of writes over their lifetimes. Of course the different database engines don't make this easy, and each has different needs. Journals, binary replication logs, error and query logs, and other miscellaneous files also require different tuning.

Before creating a dataset with a small `recordsize`, be sure you understand the interaction between VDEV type and space utilization. In some situations, disks with the smaller 512-byte sector size can provide better storage efficiency. It is entirely possible you may be better off with a separate pool specifically for your database, with the main pool for your other files.

For high-performance systems, use mirrors rather than any type of RAID-Z. Yes, for resiliency you probably want RAID-Z. Hard choices are what makes systems administration fun!

All Databases

Enabling lz4 compression on a database can, unintuitively, decrease latency. Compressed data can be read more quickly from the physical media, as there is less to read, which can result in shorter transfer times. With lz4's early abort feature, the worst case is only a few milliseconds slower than opting out of compression, but the benefits are usually quite significant. This is why ZFS uses lz4 compression for all of its own metadata and for the LZARC.

When the Compressed ARC feature lands in OpenZFS, enabling compression on the dataset will also allow more data to be kept in the ARC, the fastest ZFS cache. In a production case study done by Delphix, a database server with 768 GB of RAM went from using more than 90% of its memory to cache a database to using only 446 GB to cache 1.2 TB of compressed data. Compressing the in-memory cache resulted in a significant performance improvement. As the machine could not support any more RAM, compression was the only way to improve.

ZFS metadata can also affect databases. When a database is rapidly changing, writing out two or three copies of the metadata for each change can take up a significant number of the available IOPS of the backing storage. Normally, the quantity of metadata is relatively small compared to the



default 128 KB record size. Databases work better with small record sizes, though. Keeping three copies of the metadata can cause as much disk activity, or more, than writing actual data to the pool.

Newer versions of OpenZFS also contain a `redundant_metadata` property, which defaults to *all*. This is the original behavior from previous versions of ZFS. However, this property can also be set to *most*, which causes ZFS to reduce the number of copies of some types of metadata that it keeps.

Depending on your needs and workload, allowing the database engine to manage caching might be better. ZFS defaults to caching much or all of the data from your database in the ARC, while the database engine keeps its own cache, resulting in wasteful double caching. Setting the `primarycache` property to *metadata* rather than the default *all* tells ZFS to avoid caching actual data in the ARC. The `secondarycache` property similarly controls the L2ARC.

Depending on the access pattern and the database engine, ZFS may already be more efficient. Use a tool like `zfsmon` from the `zfs-tools` package to monitor the ARC cache hit ratio and compare it to that of the database's internal cache.

Once the Compressed ARC feature is available, it might be wise to consider reducing the size of the database's internal cache, and instead letting ZFS handle the caching. The ARC might be able to fit significantly more data in the same amount of RAM than your database can.

MySQL–InnoDB/XtraDB

InnoDB became the default storage engine in MySQL 5.5 and has significantly different characteristics than the previously used MyISAM engine. Percona's XtraDB, also used by MariaDB, is similar to InnoDB. Both InnoDB and XtraDB use a 16 KB block size, so the ZFS dataset that contains the actual data files should have its `recordsize` property set to match. We also recommend using MySQL's `innodb_one_file_per_table` setting to keep the InnoDB data for each table in a separate file rather than grouping it all into a single `ibdata` file. This makes snapshots more useful and allows more selective restoration or rollback.

Store different types of files on different datasets. The data files need 16 KB block size, lz4 compression, and reduced metadata. You might see performance gains from caching only metadata, but this also disables `prefetch`. Experiment and see how your environment behaves.

```
# zfs create -o recordsize=16k -o compress=lz4 -o
redundant_metadata=most -o primarycache=metadata mypool/var/db/mysql
```

The primary MySQL logs compress best with `gzip`, and don't need caching in memory.

```
# zfs create -o compress=gzip1 -o primarycache=none mysql/var/log/mysql
```

The replication log works best with `lz4` compression.

```
# zfs create -o compress=lz4 mypool/var/log/mysql/replication
```

Tell MySQL to use these datasets with these `/usr/local/etc/my.cnf` settings.

```
data_path=/var/db/mysql
log_path=/var/log/mysql
binlog_path=/var/log/mysql/replication
```

You can now initialize your database and start loading data.

MySQL – MyISAM

Many MySQL applications still use the older MyISAM storage engine, either because of its simplicity or just because they have not been converted to using InnoDB.

MyISAM uses an 8 KB block size. The dataset record size should be set to match. The dataset layout should otherwise be the same as for InnoDB.

PostgreSQL

ZFS can support very large and fast PostgreSQL systems if tuned properly. Don't initialize your database until you've created the needed datasets.

PostgreSQL defaults to using 8 KB storage blocks for everything. If you change PostgreSQL's block size you must change the dataset size to match.

On a default FreeBSD install PostgreSQL goes in `/usr/local/pgsql/data`. For a big install, you probably have a separate pool for that data. Here I'm using the pool `pgsql` for PostgreSQL.

```
# zfs set mountpoint=/usr/local/pgsql pgsql
# zfs create pgsql/data
```

Now we have a chicken-and-egg problem. PostgreSQL's database initialization routine expects to create its own directory tree, but we want particular subdirectories to have their own datasets. The easiest way to do this is to let PostgreSQL initialize and then create datasets and move the files.

```
# /usr/local/etc/rc.d/postgresql oneinitdb
```

The initialization routine creates databases, views, schemas, configuration files, and all the other components of a high-end database. Now you can create datasets for the special parts.

PostgreSQL stores databases in `/usr/local/pgsql/data/base`. The Write Ahead Log, or **WAL**, lives in `/usr/local/pgsql/data/pg_xlog`. Move both of these out of the way.

```
# cd /usr/local/pgsql/data
# mv base base-old
# mv pg_xlog pg_xlog-old
```

Both of these use an 8 KB block size and you would want to snapshot them separately, so create a dataset for each. As with MySQL, tell the ARC to cache only the metadata. Also tell these datasets to bias throughput over latency with the `logbias` property.

```
# zfs create -o recordsize=8k -o redundant_metadata=most -o primarycache=metadata logbias=throughput pgsql/data/pg_xlog
# zfs create -o recordsize=8k -o redundant_metadata=most -o primarycache=metadata logbias=throughput pgsql/data/base
```

Copy the contents of the original directories into the new datasets.

```
# cp -Rp base-old/* base
# cp -Rp pg_xlog-old/* pg_xlog
```

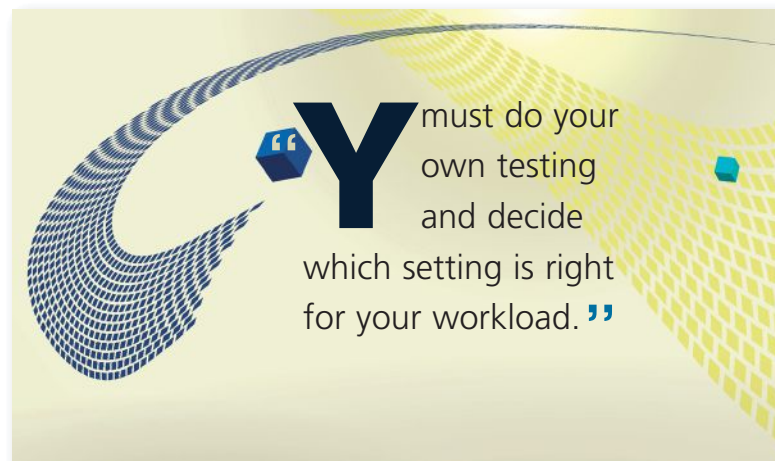
You can now start PostgreSQL.

Tuning for File Size

ZFS is designed to be a good general-purpose filesystem. If you have a ZFS system serving as file server for a typical office, you don't really have to tune for file size. If you know what size of files you're going to have, though, you can make changes to improve performance.

Small Files

When creating many small files at high speed in a system without a SLOG, ZFS spends a significant amount of time waiting for the files and metadata to finish flushing to stable storage.



You must do your own testing and decide which setting is right for your workload.”

If you are willing to risk the loss of any new files created in the last five seconds (or more if your `vfs.zfs.txg.timeout` is higher), setting the sync property to *disabled* tells ZFS to treat all writes as asynchronous. Even if an application asks that it not be told that the write is complete until the file is safe, ZFS returns immediately and writes the file along with the next regularly scheduled `txg`. A high-speed SLOG lets you store those tiny files both synchronously and quickly.

Big Files

ZFS recently added support for blocks larger than 128 KB via the `large_block` feature. If you're storing many large files, certainly consider this. The default maximum block size is 1 MB.

Theoretically, you can use block sizes larger than 1 MB. Very few systems have extensively tested this, however, and the interaction with the kernel memory allocation subsystem has not been tested under prolonged use. You can try really large record sizes, but be sure to file a bug report when everything goes sideways. The `sysctl vfs.zfs.max_recordsizes` controls the maximum block size.

Once you activate `large_blocks` (or any other feature), the pool can no longer be used by hosts that do not support the feature. Deactivate the feature by destroying any datasets that have ever had their `recordsize` set to larger than 128 KB.

Storage systems struggle to balance latency and throughput. ZFS uses the `logbias` property to decide which way it should lean. ZFS uses a `logbias` of *latency* by default, so that data is quickly synced to disk, allowing databases and other applications to continue working. When dealing with large files, changing the `logbias` property to *throughput* might result in better performance. You must do your own testing and decide which setting is right for your workload. ●

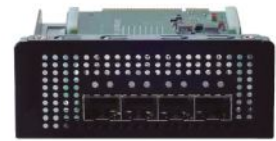
MICHAEL W. LUCAS is the author of *Absolute FreeBSD*, *Absolute OpenBSD*, and *DNSSEC Mastery*, among others. He lives in Detroit, Michigan, with his wife and a whole mess of rats. Visit his website at <https://www.michaelwlucas.com>.

ServerU

Rack-mount networking server

Designed for BSD and Linux Systems
Up to **5.5Gbit/s** routing power!

Made for  FreeBSD



PERFECT FOR

- ▶ BGP & OSPF routing
- ▶ Firewall & UTM Security Appliances
- ▶ Intrusion Detection & WAF
- ▶ CDN & Web Cache / Proxy
- ▶ E-mail Server & SMTP Filtering
- ▶ Anti-DDoS and clean pipe filtering

KEY FEATURES

- ▶ 6 NICs w/ Intel igb(4) driver w/ bypass
- ▶ Hand-picked server chipsets
- ▶ Netmap Ready (FreeBSD & pfSense)
- ▶ Up to 14 Gigabit expansion ports
- ▶ Up to 4x10GbE SFP+ expansion



Designed. Certified. Supported

| 1 Gbit/s Copper | Ports | Chipset |
|----------------------|----------------------|---------------------------|
| L800-G808-1 | 8x Gbe RJ-45 ports | 8x Intel i210 AT; PEX8618 |
| L800-G808-2 | 8x Gbe RJ-45 ports | 8x Intel i210 AT; PEX8618 |
| L800-G428-1 | 4x Gbe RJ-45 ports | 1x Intel i350 AM4 |
| L800-G428-2 | 4x Gbe RJ-45 ports | 1x Intel i350 AM4 |
| 1 Gbit/s SFP (Fiber) | Ports | Chipset |
| L800-S406-1 | 4x Gbe SFP ports | i350-AM4 |
| 10GbE Copper | Ports | Chipset |
| L800-T202-1 | 2x 10GbE RJ-45 ports | Intel X540 |
| L800-T203-1 | 2x 10GbE RJ-45 ports | Intel X540 |
| 10GbE SFP+ (Fiber) | Ports | Chipset |
| L800-X204-1 | 2x 10GbE SFP+ | Intel 82599ES |
| L800-X205-1 | 2x 10GbE SFP+ | Intel 82599ES |
| L800-X405-1 | 4x 10GbE SFP+ | Intel 82599ES; PEX8724 |

contactus@serveru.us | www.serveru.us | 8001 NW 64th St. Miami, LF 33166 | +1 (305) 421-9956