Automatic Performance Tuning in the Zettabyte File System

Val Henson
vhenson@eng.sun.com

Matt Ahrens
ahrens@eng.sun.com

Jeff Bonwick
bonwick@eng.sun.com

Sun Microsystems, Inc.
17 Network Circle
Menlo Park, CA 94025

ABSTRACT

As storage systems become ever larger and more complex, file systems and other storage software needs to move away from static configuration and manual performance tuning and towards dynamic configuration and automatic run-time performance tuning. The Zettabyte File System (ZFS) includes many self-tuning and self-managing algorithms. In this paper we present three of those algorithms: dynamic striping, automatic block size selection, and automatic filename based performance tuning.

1. INTRODUCTION

Traditionally, file systems have relied on an administrator to allocate and manage storage. Performance tuning after a file system is created is difficult or impossible without rebuilding the file system from scratch. Growing storage capacity and an increasing number of factors affecting performance make manual tuning of storage system performance a labor-intensive and inefficient process. We need new algorithms that are automatic and change dynamically with workload: in other words, software needs to tune performance itself.

In this paper, we'll describe three techniques for automatic performance tuning in the Zettabyte File System (ZFS). The first technique, dynamic disk striping, provides the performance benefit of normal disk striping without the administrative hassle. The second technique is dynamic block size selection. The third technique is a proposal for a feedback system for file system optimization using an automatic model generator to predict file size and lifetime using filenames.

2. ZFS IN A NUTSHELL

ZFS is a general purpose local file system in development at Sun Microsystems. The three main goals of ZFS are immense capacity, strong data integrity, and simple administration. To achieve the first two goals, ZFS uses 128-bit block addresses, scalable algorithms, self-validating checksums, and a copy-on-write transactional model. To simplify administration, we based ZFS around the idea of pooled storage. A storage pool is a collection of storage devices which many file systems can draw from as needed. A Storage Pool Allocator (SPA) consumes physical blocks from all the devices in a storage pool and exports virtually addressed blocks of various sizes. The SPA exports individual virtually addressed blocks, rather than exporting a single contiguous logical volume as volume managers do. The Data Management Unit (DMU) consumes the blocks exported by the SPA and exports generic objects (flat files). These objects are then used by the "file system" layer, the ZFS POSIX Layer (ZPL), to implement a standard POSIX-compliant file system.

3. DYNAMIC STRIPING

"Static striping", in which data is striped across disks in chunks of a certain size, has a number of problems. A great deal of research has gone into calculating disk striping parameters[3, 5], but the essential difficulty is that the stripe layout is statically determined at stripe group creation time, but workloads change dynamically. Some flexibility can be gained by striping the same data more than once with different widths, but even with partial coalescence of the duplicated data the storage overhead is prohibitive — more than 50%[5]. Static striping also causes the performance of the entire stripe group to be limited by the performance of the slowest disk in the group, since a streaming I/O cannot complete faster than the slowest disk can read or write the blocks stored on it[1].

Dynamic striping takes advantage of the fact that ZFS can write any block to any device in the storage pool. ZFS is a copy-on-write, write-anywhere file system: each write to a block causes the block to be written in a new location. When the SPA is presented with a sequence of blocks to write out, it automatically fans out the writes across all the available devices in the storage pool. Unlike static striping, each block in the file can be written to any location on any disk. Dynamic striping requires absolutely no administrator configuration beyond the act of creating the storage pool with the following command:

```
# zpool create my_pool disk1 disk2 ...
```

When a new device is added to the pool, the SPA begins fanning out writes to it immediately. Similarly, when a de-

vice is removed, the SPA migrates the data off the device to be removed, removing it from the "dynamic stripe group." Data is automatically redistributed across all disks as it is written. For example, if we increase the number of disks from 7 to 10, new data will be striped across all 10 disks, while old data will remain striped across the original 7. If the file system is reasonably active, the effective stripe width for reads will gradually increase from 7 to 10 over time, without any further action. Adding and removing disks can be thought of as dynamically changing the stripe width.

Dynamic striping allows ZFS to cope with heterogeneous devices, since it can allocate blocks from disks in proportion to their performance. Adding one slow device to a storage pool will not degrade performance, since ZFS can simply write fewer blocks to the slow device compared to the faster devices.

4. BLOCK SIZE SELECTION

ZFS supports multiple block sizes within a file system, from 512 bytes up to 1 MB. To do this, it divides up the available storage into metaslabs, in the style of the slab allocator[2]. By analogy to stem cells in a developing embryo, metaslabs start life as undifferentiated "stem slabs" and are divided into blocks of a certain size when they are first allocated from. ZFS automatically chooses the best block size on a per-file basis. It also allows the administrator to specify a particular block size on a per-file or per-file-system basis.

We divided applications into several classes of most commonly observed write behavior: (a) write entire file at once sequentially, (b) append slowly, (c) append quickly, (d) random record update. We observed that since ZFS is copy-on-write, we could upgrade a single-block file from one block size to the next larger block size with no penalty except when a it ends on a power of two boundary — if the file ends on any other boundary, we will have to rewrite the entire first block of the file anyway, so we might as well write it out as part of a more efficient larger block. We also decided to concentrate on optimizing block size for normal sequential write access patterns and allow record-based applications to explicitly set their own preferred block size for maximum performance, since they usually are hand-tuned for performance already (although we are working on an algorithm to automatically detect this access pattern and optimize block size for it).

Based on these observations, our algorithm for selecting block size is a function of file length. The block size is that which would result in the smallest amount of time taken to write the entire file. Using this algorithm, most files smaller than the maximum block size will use at most a few blocks. As a file grows and shrinks, its block size changes to the most optimal block size for its length. Our algorithm results in about 13% disk space waste on an NFS server used by a few hundred software engineers, which is acceptable given modern disk capacity.

5. AUTOMATIC FILENAME-BASED PERFORMANCE TUNING

Daniel Ellard, et al. have recently produced a new method of predicting file size and lifetime based on the filename chosen by an application[4]. Their system automatically generates a model without any user input. We propose extending Ellard's work by automatically rerunning the model generator whenever file system activity deviates too far from the current model. The model generator daemon would trace file system activity at a low sample rate until the workload changed, and then up the sample rate in order to generate a new model which is automatically fed back into the file system.

This system would allow automatic profiling and tuning of the file system without administrator intervention except for setting up the model generator daemon to run automatically. It throttles itself when there is no work to be done, regenerates a new model when the workload changes, and can be removed entirely if it provides no benefit.

6. CONCLUSIONS

We present three methods of automatic performance tuning that require little or no administrator effort to set up and no administrative effort to maintain. Dynamic striping provides all the performance benefits of static striping without requiring hand-tuning for specific workloads, or the rebuilding of stripe groups as workloads change or devices are added. Automatic block size selection on a per-file basis saves disk space and I/O time simultaneously without tuning block size by hand for different workloads. An automatic model generator for file system activity based on filenames could make file systems self-tuning in the face of fluctuating workloads without administrator intervention. All three techniques share the key qualities of dynamic adaptation, algorithmic simplicity, and near-zero administrative cost.

7. ADDITIONAL AUTHORS

Mark Maybee (Sun Microsystems, Inc. email: `maybee@central.sun.com`).

8. REFERENCES

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *The 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 33–40, 2001.

[2] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the 1994 USENIX Summer Technical Conference*, 1994.

[3] Peter M. Chen and David A. Patterson. Maximizing performance in a striped disk array. In *Proceedings of the 17th Annual Int'l Symp. on Computer Architecture, ACM SIGARCH Computer Architecture News*, 1990.

[4] Daniel Ellard, Jonathan Ledlie, and Margo Seltzer. The utility of file names. Technical Report TR-05-03, Computer Science Group, Harvard University, March 2003.

[5] Peter Triantafillou and Christos Faloutsos. Overlay striping and optimal parallel I/O for modern applications. *Parallel Computing*, 24(1):21–43, 1998.