Wodan: a pure OCaml, flash-aware filesystem library

Gabriel de Perthuis

June 1, 2017

Abstract

MirageOS is a library operating system that constructs unikernels for secure, high performance network applications. Until now, MirageOS lacked a reliable, usable filesystem, which limited its applicability by introducing a dependency on an external storage service. For example, the CueKeeper web application is Mirage-based and dependent on client-side storage.

We introduce Wodan, a filesystem library designed for MirageOS. It is designed with flash in mind: SSDs, NVM, hybrid devices. It provides a low-level key-value layer that Irmin (a distributed database inspired by Git) can use to store data. It is designed to be reliable and prevent bit rot. It puts control over layout details and performance trade-offs in the library user's hands.

The flash focus of the design makes for some novel design choices. To prevent wearing out the flash, the newest filesystem root can be written anywhere (in sequential order), and opening the filesystem involves bisecting for the root.

This talk will delve into the design of Wodan.

1 A functional data structure: hitchhiker trees

To make sure writes are atomic, we use a functional data structure when writing to disk. New data goes to another location instead of risking partially updating the disk. The main data structure is a hitchhiker tree, a functional variant of a B^ε-tree. B^ε-trees were introduced as buffered B-trees by Brodal and Fagerberg [BF03]. Unlike in a B-tree, which pushes data out to the leaves and has only pointers to children in parent nodes, B^ε-tree nodes contain both pointers to children and a buffer of data operations. This makes B^ε-trees much more efficient in write operations; most writes only update buffers near the root. The insert path is shorter, which means less churn, which is important because with a functional data structure modified items will be written again at a different disk location. For efficiency, in memory the data structure isn't functional, mutability is used to batch changes together and bake them into dirty nodes.

2 User-defined, flash-optimised layout

The block size should match the size of erase blocks. This prevents write amplification; smaller blocks would be write amplified by the flash translation layer, and larger blocks would amplify writes at the filesystem level by writing more data than necessary each time the data is checkpointed. This means the block size is between 256k and 4M on a standard SSD.

3 Consistency

Loss of communication with the backing device can occur at any point; the filesystem must ensure the data is in a safe state. To prevent torn writes, and detect corruption in the backing device, every block is written with a CRC32C. To prevent out of order writes, a barrier is issued prior to writing a root block, so that child block references are always valid.

4 Resiliency

The filesystem is checked at mount time. Checking the entirety of the filesystem at mount time guarantees the checker code is maintained and matches every implemented feature. It also allows early warning of bit rot. Finally, this scan provides an opportunity to build data structures such as a bitmap of in-use blocks.

5 User control

The data layout (block size, key size) is parameterised using OCaml functors. Key operations such as flushing are user-controlled; this makes performance more predictable.

6 Fast mounting

Mount time checks can be expensive. For large filesystems, they can be made optional by logging the in-use bitmap periodically. An up to date bitmap can be reconstructed from a slightly stale bitmap and the log of newer write operations. This bitmap is stored with a well-known key which is defined in the super block.

7 Testing

The filesystem is packaged as a library, built with jbuilder. A first Mirage unikernel exercises the filesystem code by doing writes of random data, periodically closing and reopening the filesystem. A second unikernel is instrumented with American Fuzzy Lop and does a single insert into a fuzzer-generated hostile image. AFL looks at the branches the code takes to maximise coverage. Fuzzing involves magic CRCs that are regenerated in the test harness; AFL's genetic algorithm would otherwise be unable to generate valid CRCs most of the time.

References

[BF03] Gerth Stolting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 546–554, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.