Controlling File System Write Ordering

Nathan C. Burnett, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
Computer Sciences Department, University of Wisconsin – Madison

1 Introduction

Traditional operating system kernels present a relatively narrow interface to applications for I/O. This interface typically consists of `open()`, `close()`, `read()`, `write()`, `lseek()`and `fsync()`. Some applications, however, require more control than this limited interface allows. For example, a database management system such as Oracle or PostgreSQL, performs write-ahead logging to provide transactional semantics to users. In order to ensure that the database can be brought to a consistent state in the event of a crash, the DBMS must ensure that updates to the log are committed to disk strictly before updates to the data itself. So, the DBMS needs to control the order in which data is committed to stable storage for correctness.

In current operating systems, there are essentially two ways to control disk write ordering. In the first, the application accesses the raw storage device, bypassing the filesystem and kernel buffer cache. This provides the desired control, but at the cost of application portability, complexity and system managability. The second way is to use synchronous I/O, such as `fsync()`. This also provides the required control, but this time at the cost of performance. A third, degenerate, option is to forego controlling write ordering entirely. This solution is fast, simple and portable, but can be disastrous in the event of a system crash. Any application with complex, on-disk data structures will have a similar set of trade-offs.

In this work, we seek to provide a new interface which allows the application to control the order in which data is written to disk. This interface should be fast (*i.e.* as asynchronous as possible), and simple enough that it will be easy to standardize to facilitate portability.

2 Approach

Our approach is to allow an application to specify ordering dependencies between write operations. We propose two alternative methods for allowing this type of control, *file system barriers* and *asynchronous graphs*.

2.1 File System Barriers

File system barriers entail the addition of a new system call, `barrier()`. This call guarantees that the data from any `write()` calls made *previous* to the `barrier()` call will be commited to stable storage strictly *before* the data from any subsequent calls to *write*. `barrier()` is global in scope, that is, the ordering is imposed on all write operations, without regard to which process requested them. This interface makes it easy to convert existing applications to use the new interface, simply replace calls to `fsync()` and `sync()` with calls to `barrier()`. For example, a DBMS would call `write()` to update the write-ahead log, call `barrier()` and finally call `write()` to update the data itself. This ensures that the on-disk log will be updated before the on-disk data is updated.

We have implemented file system barriers in FreeBSD 5.4. Unfortunately, we found that if the workload normally issues only a few writes between `barrier()` calls, the disk scheduler and buffer cache manager become overly constrained and performance suffers.

2.2 Asynchronous Graphs

After our experience with file system barriers, we realized that the application needed to be able to specify ordering constraints at a finer granularity than barriers allowed. Thus, we propose *asynchronous graphs*. In this scheme, each time an application calls `write()` the kernel assigns a unique identifier to that write operation and returns it to the application. When making subsequent calls to `write()`, the application can use these identifiers to inform the kernel that the data from the latest `write()` operation should be committed to stable storage only *after* the data from the specified writes. This allows an application to express ordering constraints only where ordering matters, instead of imposing global constraints on all of the writes in the system. The kernel is free to reorder writes arbitrarily if no ordering constraint was specified. This gives the application the power to specify its ordering requirements, while allowing the operating system to use its normal I/O optimizations for all other I/O.

Using our example of a DBMS again, the DBMS calls `write()` to update the log, and is given an identifier for that write. It then calls `write()` a number of times to update the data, each time passing in the identifier for the log write to ensure that the data updates are written to disk after the log updates.

3 Current Status

So far, we have been exploring the asynchronous graph concept within a simple simulator. Our simulator simulates the buffer cache itself, and distinguishes between sequential and non-sequential disk accesses. That is, sequential accesses are fast and non-sequential accesses are slow. So far we have been able to show that asynchronous graphs require fewer writes and, in particular, fewer non-sequential writes than using fsync or filesystem barriers for ordering with a TPC-B-like workload.

In the near future we are extending our simulator to take into account clustered writes, the buffer cleaning daemon and a more accurate disk model. In the following months, we plan to implement the asynchronous graph interface in FreeBSD and evaluate its performance on real and synthetic workloads.