Using Berkeley DB in the Linux Kernel

Aditya Kashyap, Jay Dave, Harikesavan Krishnan, Charles P. Wright,
Mohammad Nayyer Zubair, and Erez Zadok
Stony Brook University

1 Introduction

Computers need to manage data efficiently. In user space there is an abundance of solutions to address this problem: simple configuration files, embedded databases, and full-blown relational databases. The kernel also needs to manage data, and it would benefit from having an efficient and reliable database system.

We have ported the Berkeley Database (BDB) [3] into the Linux kernel (we call our port KBDB). BDB provides the ability to efficiently and persistently store key-value pairs using hash tables, B+-trees, queues, and indexed by a logical record number. BDB is a stable, full-featured, and widely-used embedded database with support for transactions and replication.

There are many possible applications for an in-kernel database. Extended Attributes and ACLs could easily be added to existing on-disk file systems. Rather than changing the existing format of a file system, the attributes could be stored in a database. An in-kernel intrusion detection system could store usage statistics and security policies within a database. As the database provides a backing store, if the system is brought down, the data will be committed to disk rather than being lost. As a sample application, we have implemented a kernel file system based on Berkeley DB (KBDBFS). Using KBDB, three graduate students developed this file system in less than one month. We have shown that our file system has acceptable overheads over Ext2.

2 BDB Porting Considerations

Figure 1 shows the components that we identified and ported to the kernel BDB package. The complexity of porting ranged from trivial to complex code changes. We had to modify most of the components to address the trivial porting issues in code like changing user-level function calls to their corresponding kernel equivalents. Complex changes to the code involved ensuring integrity with concurrent accesses to the database by more than one program, and supporting `mmap` at the kernel level. BDB allows multiple processes to access the same database. In user-space this is done with shared memory. Processes, however, do not exist in the kernel and there is only one (physical) address space. Shared memory concepts in user-land had to be carefully examined in the context of the kernel address space.

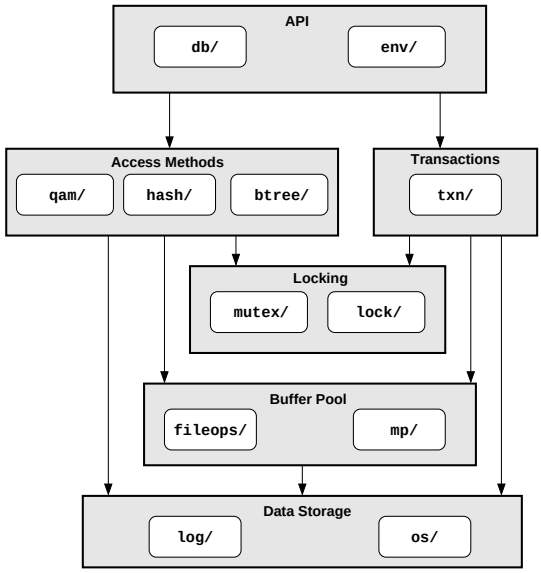

Figure 1: In-Kernel Berkeley DB Architecture

3 KBDBFS

An in-kernel database can be useful for a variety of applications because of its flexibility, performance, reliability, and scalability features. One obvious application of an in-kernel database is a file system that uses a database as a backing store. We have designed a simple file system that we call KBDBFS, to demonstrate the utility of Berkeley DB in the kernel. KBDBFS supports simple file system operations like `create`, `unlink`, `readdir`, `read`, and `write`, as well as `mmap`-ed operations.

Figure 2 presents an architectural overview of KBDBFS. The file system contains two primary components: the Berkeley DB kernel module, and the file system module. KBDBFS uses three databases: `dirent.db` maps file names to inode numbers; `metadata.db` maps inode numbers to STAT information; and `data.db` maps pages within a file to data. Like other file systems, KBDBFS uses the page and inode caches to store information before writing it to the backing store. KBDBFS has two modes of file `write` operations, asynchronous and synchronous.

Evaluation We compared the performance of KBDBFS, Ext2, and Ext2 running on a loopback device. We used a loopback device for Ext2 because it double buffers like KBDBFS (KBDBFS buffers data

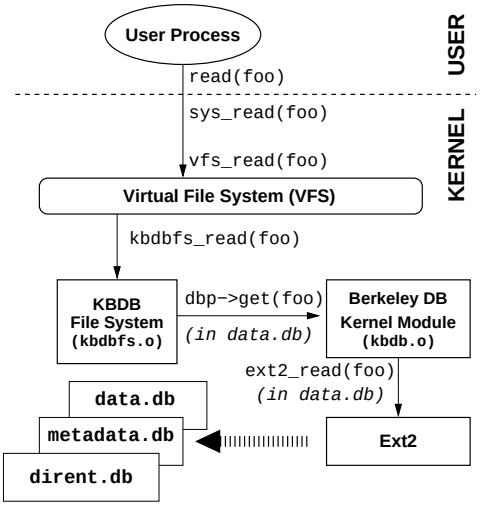

Figure 2: KBDBFS Architecture

and Ext2 buffers the database).

We conducted all tests on a Pentium-IV 1.7GHz with 1GB of RAM. The machine ran Red Hat Linux 9 and a vanilla 2.4.22 kernel. To ensure a cold cache, we unmounted the underlying file system once after loading the test data. For all tests, we computed 95% confidence intervals for the mean elapsed, system, and user time.

We report here the results from Bonnie, an I/O intensive benchmark [1].

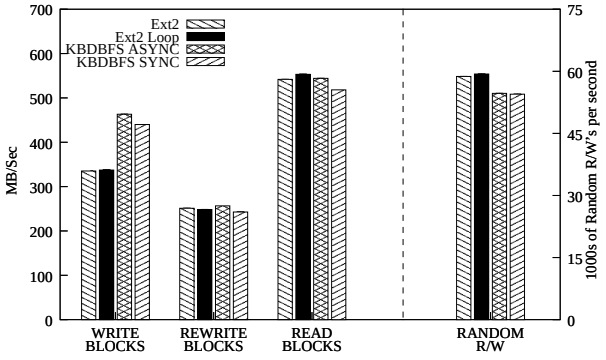

Figure 3: Bonnie with a 16MB file. The right Y axis applies only to the Random R/W bars.

Figure 3 shows the bytes processed per second during sequential block writes, block re-writes, and block reads. The second part of the graph shows the number of random reads and writes per second. KBDBFS uses the page cache, so heavy writers are not throttled as in case of Ext2 which uses the buffer cache. This is why KBDBFS mounted in ASYNC mode has a 38% higher bandwidth than Ext2, and KBDBFS mounted in SYNC mode has a 31% higher bandwidth. While re-writing, file systems have to read-in the page into memory and then eventually write it out. Thus we see a sharp drop in the bandwidth for each of the file systems. Here, also, KBDBFS in ASYNC mode has a 2% higher bandwidth than Ext2, but in SYNC mode, KBDBFS is nominally slower, as dirty pages might have been flushed to disk. This is also why KBDBFS in SYNC mode has a 4.4% lesser bandwidth than Ext2 for block read, while KBDBFS in ASYNC mode and the loopback file system have 0.4% and 3.7% higher bandwidths, respectively. KBDBFS in ASYNC and SYNC mode is 7% and 7.2% (respectively) slower than Ext2 when performing random seeks, while the loopback file system is 1% faster.

We also benchmarked KBDBFS under Postmark, another I/O intensive tool [2]. Our measurements include the times to unmount the file system, since KBDBFS in ASYNC mode flushes out most of the dirty data at unmount time, which needs to be accounted for. We found that KBDBFS, mounted with the SYNC mount time option has an overhead of 9.9% over Ext2 while ASYNC mount time option causes KBDBFS to have an overhead of 27.4%.

4 Conclusion

The Berkeley DB (BDB) is a powerful database that is also compact and efficient, making it suitable for embedded environments such as operating systems. BDB offers several efficient access methods, a well-defined and easy-to-use API, backing store support, transactions and more. Our work, KBDB, is the first port of such an efficient database to the Linux kernel. With KBDB, many new efficient kernel services can be rapidly developed.

As a demonstration of the utility of KBDB, we developed KBDBFS, a kernel-level file system that uses KBDB. KBDBFS is the first in-kernel file system ever developed using an in-kernel database. KBDBFS supports many traditional file system features, runs efficiently in the kernel, and was developed rapidly thanks to the availability of KBDB. Our performance benchmarks show that KBDBFS, while developed in a fraction of the time that Ext2 was, rivals Ext2's performance.

We are currently enganged in several other KBDB-based projects, including a fully POSIX compliant file system, Extended Attribute augmentation, storing secure NFS file handles persistently, a registry-like kernel configuration, and more.

References

[1] R. Coker. The Bonnie home page. `www.textuality.com/bonnie`, 1996.

[2] J. Katcher. PostMark: a New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[3] M. Seltzer and O. Yigit. A new hashing package for UNIX. In *Proceedings of the Winter USENIX Technical Conference*, pages 173–84, January 1991. `www.sleepycat.com`.